

DATA STRUCTURES USING C

Module 2 - Lecture III

STACKS AND QUEUES

Prepared By
Ms. Neetu Narayan

Stack Objectives:

- Impart in-depth knowledge of data structure and its implementation in computer programs.
- Make students understand the concepts of Stack and Queue linear data structure.
- Illustrate asymptotic notations and their usage.

3. Evaluation of Postfix expression

- Read all the symbols one by one from left to right in the given Postfix Expression.
- If the reading symbol is operand, then push it on to the Stack.
- If the reading symbol is operator (+ , - , * , / etc.,), then perform two pop operations and store the two popped operands in two different variables (operand1(stack top symbol) and operand2(next-to-top symbol)). Then perform reading symbol operation using operand2 operator operand1 and push result back on to the Stack.

Example: A B * C +

Reading Symbol	Stack operation	Stack Content
A	Push	A
B	Push	A,B
*	Operand1=B Operand2=A A*B	(A*B)
C	Push	(A*B), C
+	Operand 1=C Operand2=(A*B)	(A*B)+C

Example: Evaluate $(5+6*3)+(5+6)^3$ using Postfix Notation

Infix expression: $(5+6*3)+(5+6)^3$

Current Symbol	Stack	Postfix Expression
(((
5	((5
+	((+	5
6	((+	5 6
*	((+ *	5 6
3	((+ *	5 6 3
)	(5 6 3 * +
+	(+	5 6 3 * +
((+ (5 6 3 * +
5	(+ (5 6 3 * + 5
+	(+ (+	5 6 3 * + 5

Priority Level
1. +, -
2. *, /
3. ^
4. ++, --

Infix expression: $(5+6*3)+((5+6)*3 = 56)$

Current Symbol	Stack	Postfix Expression
6	(+ (+	5 6 3 * + 5 6
)	(+	5 6 3 * + 5 6 +
*	(+ *	5 6 3 * + 5 6 +
3	(+ *	5 6 3 * + 5 6 + 3
)		5 6 3 * + 5 6 + 3 * +

Priority Level
1. +, -
2. *, /
3. ^
4. ++, --

Postfix Evaluation of Expression: 5 6 3 * + 5 6 + 3 * +

Reading Symbol	Stack operation	Stack Content
5	Push	5
6	Push	5 6
3	Push	5 6 3
*	Pop3,Pop 6 and push 6*3	5 18
+	Pop18,Pop 5 and Push 5+18	23
5	Push	23 5
6	Push	23 5 6
+	Pop6,Pop 5 and push 5+6	23 11
3	Push	23 11 3
*	Pop3,Pop 11 and Push 11*3	23 33
+	Pop33, Pop 23 and Push 23+33	56

Evaluate: ABC+*CBA-+* where A=1, B=2, C =3.

S.N	Scan char	Value	Op1	Op2	Result	Vstack
1	A	1				1
2	B	2				1, 2
3	C	3				1,2,3
4	+		2	3	5	1,5
5	*		1	5	5	5
6	C	3				5,3
7	B	2				5,3,2
8	A	1				5,3,2,1
9	-		2	1	1	5,3,1
10	+		3	1	4	5,4
11	*		5	4	20	20
Final Result of Expression : 20						

4. Conversion from infix to prefix expression

Read all the symbols one by one from right to left in the given Infix Expression.

1. Push ')' onto Stack and '(' at the end of the infix expression.
2. If the symbol is an operand, then directly add it to output expression.
3. If the reading symbol is right parenthesis ')', then Push it on to the Stack.
4. If the reading symbol is LEFT parenthesis '(', then pop all the symbols from the stack until a RIGHT parenthesis appears. Discard the right parenthesis and add remaining popped symbols to the output expression in the order in which they are popped.
5. If the reading symbol is an operator (+ , - , * , / etc.,), then Check, if the operator on the top of the stack has higher precedence than the one being read, pop the operator and add it to the output expression. Repeat the process until a lower or equal precedence operator appears at the top of the stack. Then push the current operator onto the stack.
6. At the end after reading the entire infix expression, reverse the output expression.

Reverse of output expression is the Prefix expression.

EXAMPLE: $A^*B+C \rightarrow (A^*B+C)$

current symbol	operator stack	Output Expression
C)	C
+) +	C
B) +	C B
*) + *	C B
A) + *	C B A
(C B A * +
INFIX EXPRESSION= REVERSE OF OUTPUT EXPRESSION=		
+ * ABC		

- Convert to PREFIX expression

$$(A-B/C) * (A/K-L)$$

$(A-B/C) * (A/K-L)$		
))	
)))	
L))	L
-))-	L
K))-	LK
/))-/	LK
A))-/	LKA
()	LKA/-
*)*	LKA/-
))*)	LKA/-
C)*)	LKA/-C
/)*)/	LKA/-C
B)*)/	LKA/-CB
-)*)-	LKA/-CB/
A)*)-	LKA/-CB/A
()*	LKA/-CB/A-
(LKA/-CB/A-*

Ans: *-A/BC-/AKL

***-A/BC-/AKL**

5. Evaluation of Prefix expression

- Read all the symbols one by one from right to left in the given Prefix Expression.
- If the reading symbol is operand, then push it on to the Stack.
- If the reading symbol is operator (+ , - , * , / etc.,), then perform two pop operations and store the two popped operands in two different variables (operand1(stack top symbol) and operand2(next-to-top symbol)). Then perform reading symbol operation using operand1 operator operand2 and push result back on to the Stack.

Example: +*ABC

Reading Symbol	Stack operation	Stack Content
C	Push	C
B	Push	C B
A	Push	C B A
*	Pop A, Pop B, Push (A*B)	C (A*B)
+	Pop (A*B), POP C, Push(A*B)+C	(A*B)+C

Example: Evaluate $(5+6*3)+(5+6)^3$ using Prefix Notation

Infix expression: $((5+6*3)+(5+6)^3$

Current Symbol	Stack	Output Expression
3)	3
*) *	3
)) *)	3
6) *)	3 6
+) *) +	3 6
5) *) +	3 6 5
() *	3 6 5 +
+) +	3 6 5 + *
)) +)	3 6 5 + *
3) +)	3 6 5 + * 3
*) +) *	3 6 5 + * 3

Infix expression: $((5+6*3)+(5+6)^3$

Current Symbol	Stack	Output Expression
6) +) *	3 6 5 + * 3 6
+) +) +	3 6 5 + * 3 6 *
5) +) +	3 6 5 + * 3 6 * 5
() +	3 6 5 + * 3 6 * 5 +
(3 6 5 + * 3 6 * 5 ++
Prefix Expression= Reverse of Output Expression= + + 5 * 6 3 * + 5 6 3		

Prefix Evaluation of Expression: + + 5 * 6 3 * + 5 6 3

Reading Symbol	Stack operation	Stack Content
3	Push	3
6	Push	3 6
5	Push	3 6 5
+	Pop 5, Pop 6, Push(5+6)	3 11
*	Pop11, Pop3, Push(11*3)	33
3	Push	33 3
6	Push	33 3 6
*	Pop 6, Pop3, Push(6*3)	33 18
5	Push	33 18 5
+	Pop 5, Pop 18, Push(5+18)	33 23
+	Pop 23, Pop33, Push(33+23)	56

Example: $- * + 4 3 2 5$

Symbol	opnd1	opnd2	value	opndstack
5				5
2				5, 2
3				5, 2, 3
4				5, 2, 3, 4
+	4	3	7	5, 2
*	7	2	14	5
-	14	5	9	5, 14
				9

result



DATA STRUCTURES USING C

Module 2 - Lecture II STACKS AND QUEUES

Prepared By
Ms. Neetu Narayan
Edited by
Ms. Smriti Sehgal

Stack Objectives:

- Impart in-depth knowledge of data structure and its implementation in computer programs.
- Make students understand the concepts of Stack and Queue linear data structure.
- Illustrate asymptotic notations and their usage.

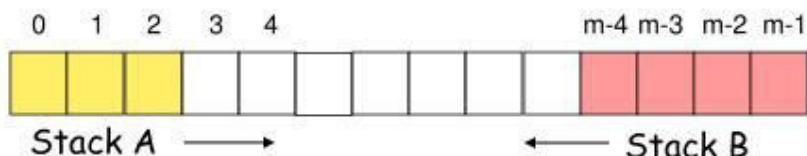
Multiple Stacks

- In stack using arrays, size is to be given at the time of declaration.
 - If array is allocated less space, frequent OVERFLOW conditions occurs
 - If array is allocated sufficiently large space, space may remain unused.

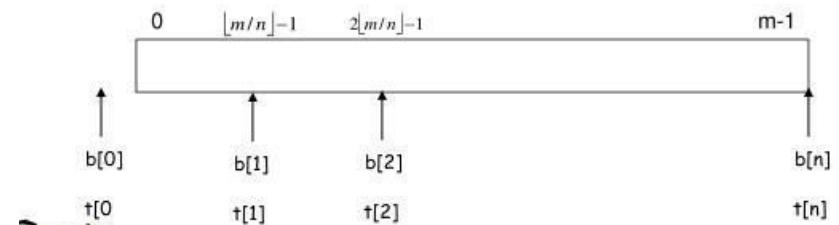
Solution: Create a large single array in which many stacks can reside, known as multiple stacks.

Multiple Stack Array

Two Stack Array



n Stack Array



Applications of Stacks

1. Parentheses Checker
2. Conversion from infix to postfix expression
3. Evaluation of Postfix expression
4. Conversion from infix to prefix expression
5. Evaluation of Prefix expression
6. Recursion
7. Tower of Hanoi

1. Parentheses Checker

Algorithm:

1. Whenever we see an opening parenthesis, we put it on stack.
2. For closing parenthesis, check what is at the top of the stack, if it corresponding opening parenthesis, remove it from the top of the stack.
3. If parenthesis at the top of the stack is not corresponding opening parenthesis, return false, as there is no point check the string further.
4. After processing entire string, check if stack is empty or not.
 - 4.a If the stack is empty, return true.
 - 4.b If stack is not empty, parenthesis do not match

$\{()\[(([]))]\}$

Input	Character in consideration	Operation	Stack Symbol
{()	{	PUSH	{
)	(PUSH	{ (
))	POP ({
[[PUSH	{ [
((PUSH	{ [(
[[PUSH	{ [([
)]	POP [{ [(
))	POP ({ [
]]	POP [{
}	}	POP {	Empty
		Accepted	

Input	Character in consideration	Operation	Stack Symbol
{()()	{	PUSH	{
)()	(PUSH	{(
)())	POP ({
{	(PUSH	{(
}	}	POP { (but couldnot find '{')	{(
Error Not Balanced			

Expression

An expression is a collection of operators and operands that represents a specific value.

Based on the operator position, expressions are divided into three categories. They are as follows...

- Infix Expression
- Postfix Expression (Reverse Polish)
- Prefix Expression (Polish)

Infix Expression

- Operator is used in between operands.

Operand1 Operator Operand2

- Example : **a+b**

Postfix Expression

- Operator is used after operands.

Operand1 Operand2 Operator

- Example : **ab+**

Prefix Expression

- operator is used before operands.

Operator Operand1 Operand2

- Example : **+ab**

2. Conversion from infix to postfix expression

Read all the symbols one by one from left to right in the given Infix Expression.

1. Push '(' onto Stack and ')' at the end of the infix expression.
2. If the symbol is an operand, then directly add it to final postfix expression(Output).
3. If the reading symbol is left parenthesis '(', then Push it on to the Stack.
4. If the reading symbol is right parenthesis ')', then pop all the symbols from the stack until a left parenthesis appears. Discard the left parenthesis and add remaining popped symbols to the postfix expression in the order in which they are popped.
5. If the reading symbol is an operator (+ , - , * , / etc.,), then Check,
 1. if the operator on the top of the stack has higher or equal precedence than the one being read, pop the operator and add it to the postfix expression.
 2. Repeat the process until a lower precedence operator appears at the top of the stack.
 3. Then push the current operator onto the stack.

Infix to Postfix

Infix Expression= **A*B+C**

current symbol	Stack	postfix string
A	(A
*	(*	A
B	(*	A B
+	(+	A B * {pop and print the '*' before pushing the '+'}
C	(+	A B * C
)		A B * C +

* Has higher precedence than +

Practice Eg.

1. $(A-B) + C * D / E - C$

2. $(A - 2 * (B + C) / D * E) + F$

3. $A + B * C / D - F + A^E$

SOLUTION

$$1. \ A B - C D * E / + C -$$

$$2. \ A^2 B C + * D / E * - F +$$

$$3. \ A B C * D / + F - A E ^ +$$



DATA STRUCTURES USING C

Module 2 - Lecture IV

STACKS AND QUEUES

Prepared By
Ms. Neetu Narayan

Stack Objectives:

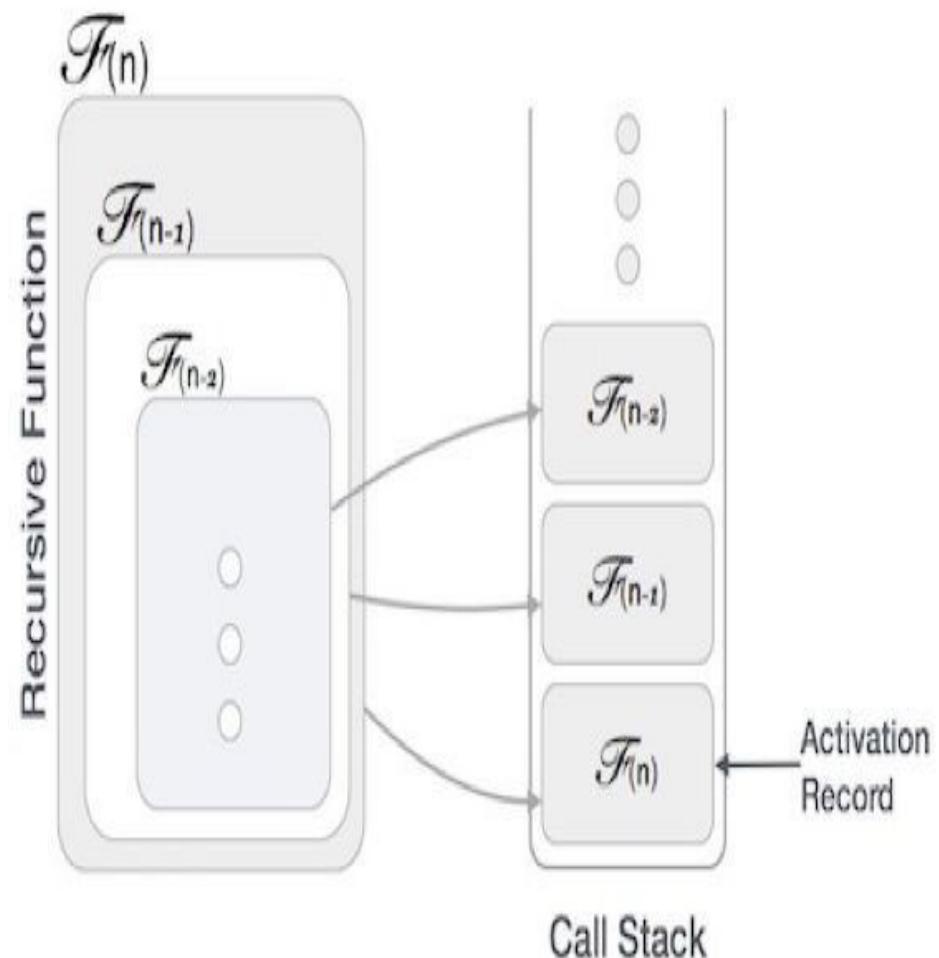
- Impart in-depth knowledge of data structure and its implementation in computer programs.
- Make students understand the concepts of Stack and Queue linear data structure.
- Illustrate asymptotic notations and their usage.

6. Recursion

Recursion is the process by which function calls itself repeatedly, until the specified condition has been satisfied.

To solve a problem recursively, two conditions must be satisfied:

1. A base value should be defined for which the function should not call itself.
2. At each function call, the argument of the function should move close to the base value.



Factorial of a number using recursion

Algorithm:

FACTO (FAC, NUMBER)

1. If NUMBER=0,
then FAC=1 and Return
2. FACTO (FAC, NUMBER-1)
3. FAC=NUMBER*FAC
4. Return

FACTO (1, 4)- Input

FACTO (1,3)

FACTO (1,2)

FACTO (1,1)

FACTO(1,0)

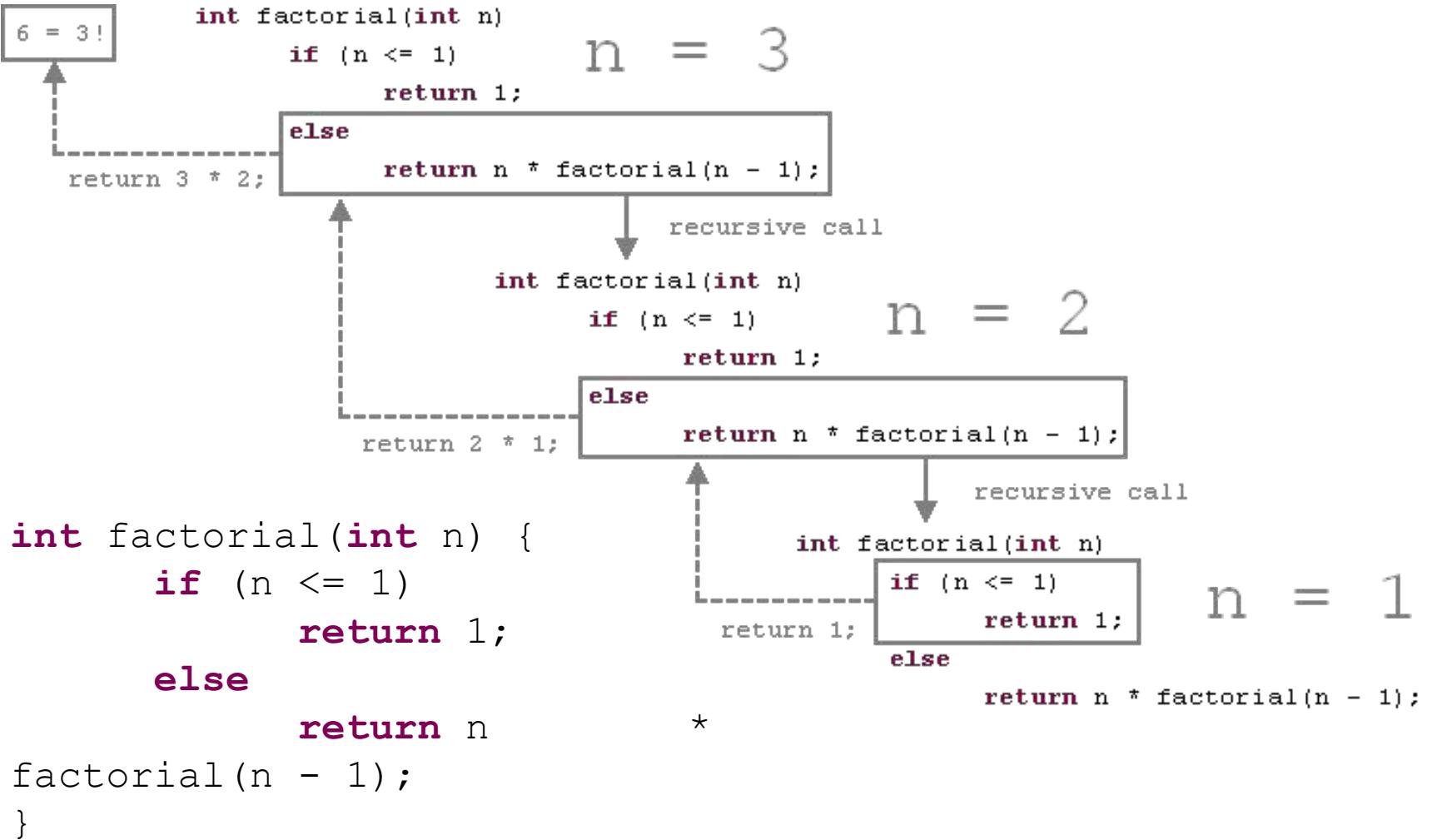
FAC=1

FACTO(1,1)=1*FACTO (1,0)=1

FACTO(1,2)=2*FACTO(1,1)=2

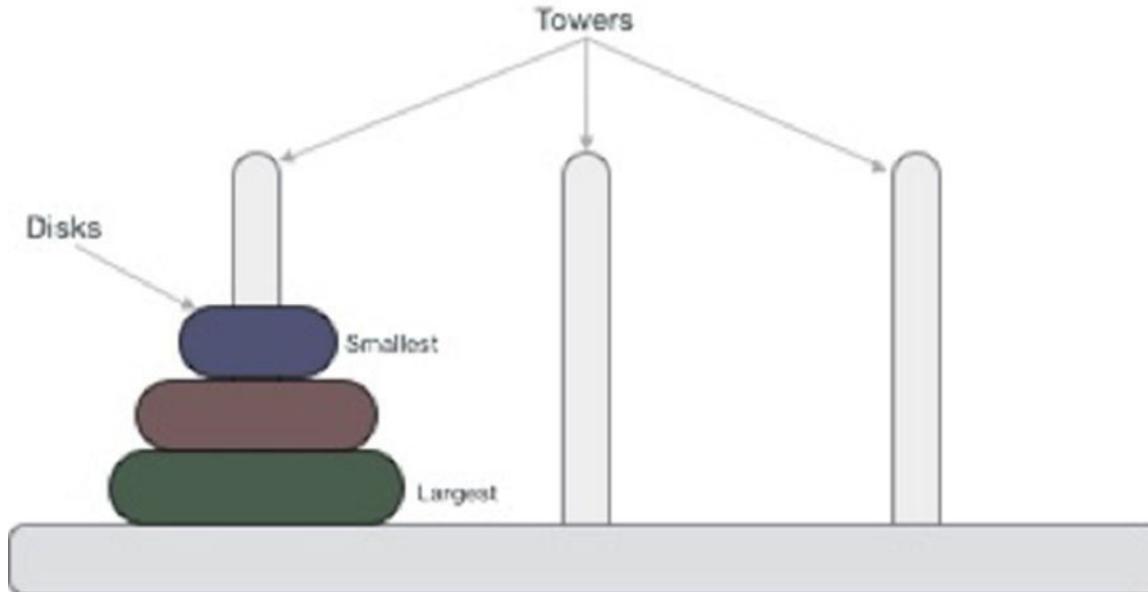
FAC= 3*FACTO(1,2)=6

FAC=4*FACTO(1,4)=24



7. Tower of Hanoi

- Mathematical puzzle
- Consists of three tower (pegs) and more than one rings;
- These rings are of different sizes and stacked upon in ascending order



- There are other variations of puzzle where the number of disks increase, but the tower count remains the same.

Rules

The mission is to move all the disks to some another tower without violating the sequence of arrangement.

1. Only one disk can be moved among the towers at any given time.
2. Only the "top" disk can be removed.
3. No large disk can sit over a small disk.

Algorithm for Towers of Hanoi

START

Procedure Hanoi(disk, source, dest, aux)

 IF disk == 1, THEN

 move disk from source to dest

 ELSE

 Hanoi(disk - 1, source, aux, dest) // Step 1

 move disk from source to dest // Step 2

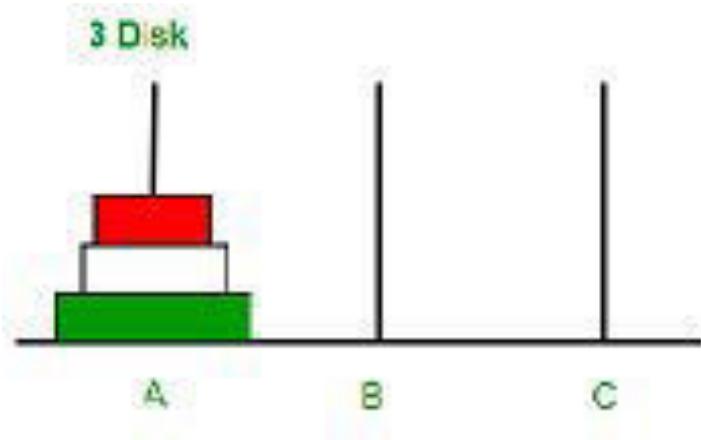
 Hanoi(disk - 1, aux, dest, source) // Step 3

 END IF

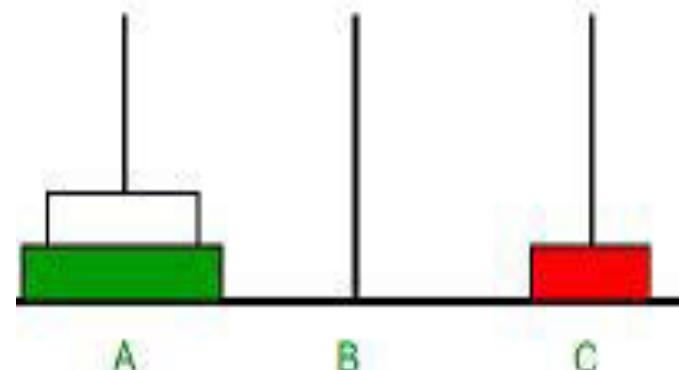
END Procedure

STOP

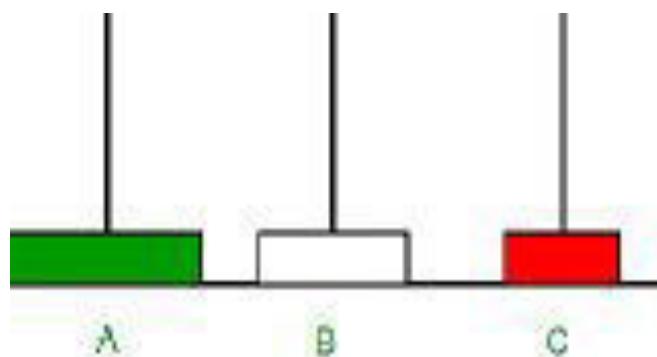
Example of Tower of Hanoi for disk=3



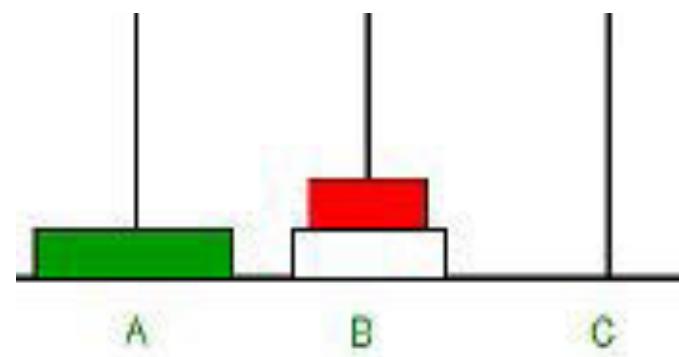
Initial



A->C

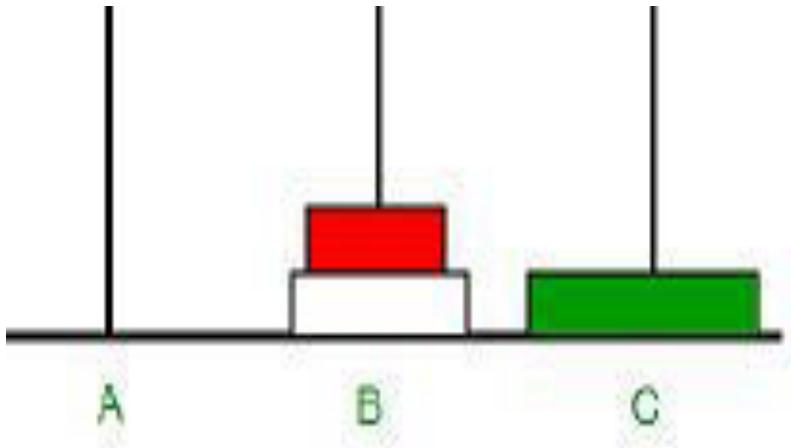


A->B

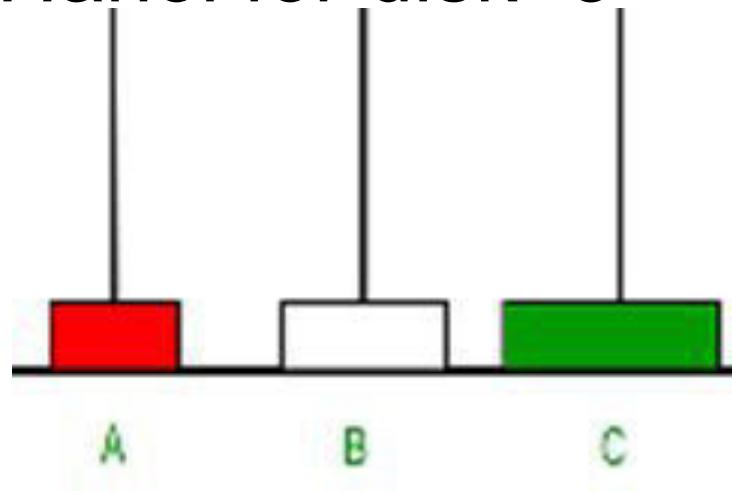


C->B

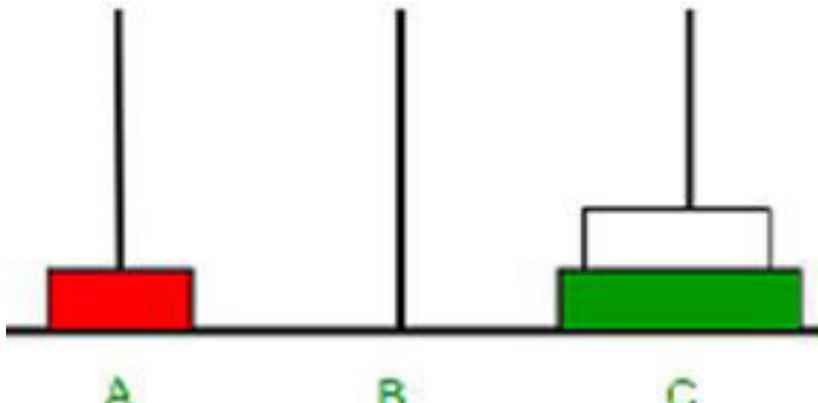
Example of Tower of Hanoi for disk=3



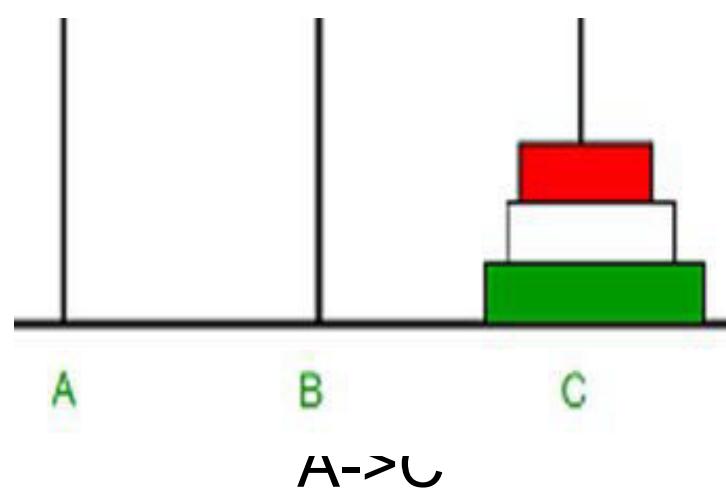
A->C



B->A



B->C



A->C

Video References

<http://towersofhanoi.info/Animate.aspx>

<http://britton.disted.camosun.bc.ca/hanoi.swf>

- Convert from infix to postfix & Prefix expression:
 - $((A-B)+D)/((E+F)^*G))$
 - $14/7^*3-4+9/2$
- Convert from prefix to infix expression:
 - $*-+ABCD$
 - $+-a^*BCD$

Practice Question

Amity School of Engineering & Technology

- Write a function that accepts 2 stacks.
Copy the contents of one stack to another. Note that the order of the elements must be preserved.



DATA STRUCTURES USING C

Module 2 - Lecture II STACKS AND QUEUES

Prepared By
Ms. Neetu Narayan
Edited by
Ms. Smriti Sehgal

Stack Objectives:

- Impart in-depth knowledge of data structure and its implementation in computer programs.
- Make students understand the concepts of Stack and Queue linear data structure.
- Illustrate asymptotic notations and their usage.

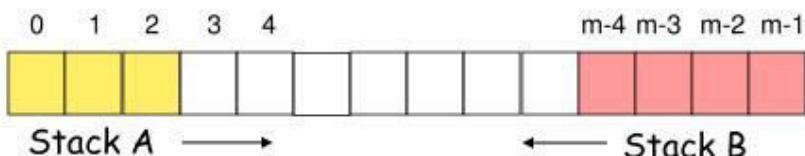
Multiple Stacks

- In stack using arrays, size is to be given at the time of declaration.
 - If array is allocated less space, frequent OVERFLOW conditions occurs
 - If array is allocated sufficiently large space, space may remain unused.

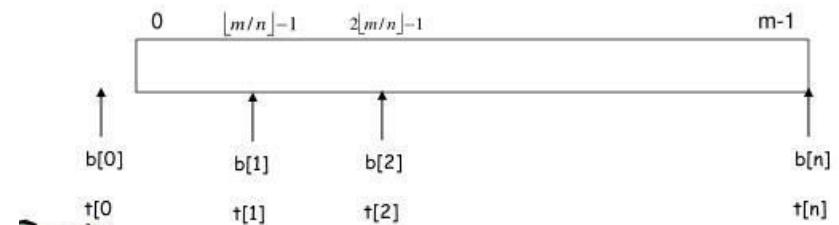
Solution: Create a large single array in which many stacks can reside, known as multiple stacks.

Multiple Stack Array

Two Stack Array



n Stack Array



Applications of Stacks

1. Parentheses Checker
2. Conversion from infix to postfix expression
3. Evaluation of Postfix expression
4. Conversion from infix to prefix expression
5. Evaluation of Prefix expression
6. Recursion
7. Tower of Hanoi

1. Parentheses Checker

Algorithm:

1. Whenever we see an opening parenthesis, we put it on stack.
2. For closing parenthesis, check what is at the top of the stack, if it corresponding opening parenthesis, remove it from the top of the stack.
3. If parenthesis at the top of the stack is not corresponding opening parenthesis, return false, as there is no point check the string further.
4. After processing entire string, check if stack is empty or not.
 - 4.a If the stack is empty, return true.
 - 4.b If stack is not empty, parenthesis do not match

$\{()\[(([]))]\}$

Input	Character in consideration	Operation	Stack Symbol
{()	{	PUSH	{
)	(PUSH	{ (
))	POP ({
[[PUSH	{ [
((PUSH	{ [(
[[PUSH	{ [([
])]	POP [{ [(
))	POP ({ [
]]	POP [{
}	}	POP {	Empty
		Accepted	

Input	Character in consideration	Operation	Stack Symbol
{()()	{	PUSH	{
)()	(PUSH	{(
)())	POP ({
{	(PUSH	{(
}	}	POP { (but couldnot find '{')	{(
Error Not Balanced			

Expression

An expression is a collection of operators and operands that represents a specific value.

Based on the operator position, expressions are divided into three categories. They are as follows...

- Infix Expression
- Postfix Expression (Reverse Polish)
- Prefix Expression (Polish)

Infix Expression

- Operator is used in between operands.

Operand1 Operator Operand2

- Example : **a+b**

Postfix Expression

- Operator is used after operands.

Operand1 Operand2 Operator

- Example : **ab+**

Prefix Expression

- operator is used before operands.

Operator Operand1 Operand2

- Example : **+ab**

2. Conversion from infix to postfix expression

Read all the symbols one by one from left to right in the given Infix Expression.

1. Push '(' onto Stack and ')' at the end of the infix expression.
2. If the symbol is an operand, then directly add it to final postfix expression(Output).
3. If the reading symbol is left parenthesis '(', then Push it on to the Stack.
4. If the reading symbol is right parenthesis ')', then pop all the symbols from the stack until a left parenthesis appears. Discard the left parenthesis and add remaining popped symbols to the postfix expression in the order in which they are popped.
5. If the reading symbol is an operator (+ , - , * , / etc.,), then Check,
 1. if the operator on the top of the stack has higher or equal precedence than the one being read, pop the operator and add it to the postfix expression.
 2. Repeat the process until a lower precedence operator appears at the top of the stack.
 3. Then push the current operator onto the stack.

Infix to Postfix

Infix Expression= **A*B+C**

current symbol	Stack	postfix string
A	(A
*	(*	A
B	(*	A B
+	(+	A B * {pop and print the '*' before pushing the '+'}
C	(+	A B * C
)		A B * C +

* Has higher precedence than +

Practice Eg.

1. $(A-B) + C * D / E - C$

2. $(A - 2 * (B + C) / D * E) + F$

3. $A + B * C / D - F + A^E$

SOLUTION

$$1. \ A B - C D * E / + C -$$

$$2. \ A^2 B C + * D / E * - F +$$

$$3. \ A B C * D / + F - A E ^ +$$



Data Structures Using C

Course Code: CSIT124

B. Tech. (IT/CSE)

Module II - Lecture 1

Made by:

Smriti Sehgal

Contents to be Covered

- **Stack:**
- Definition
- Stack Representation
 - Array Representation
 - Linked List Representation
- Operations on Stacks- Push & Pop

STACK

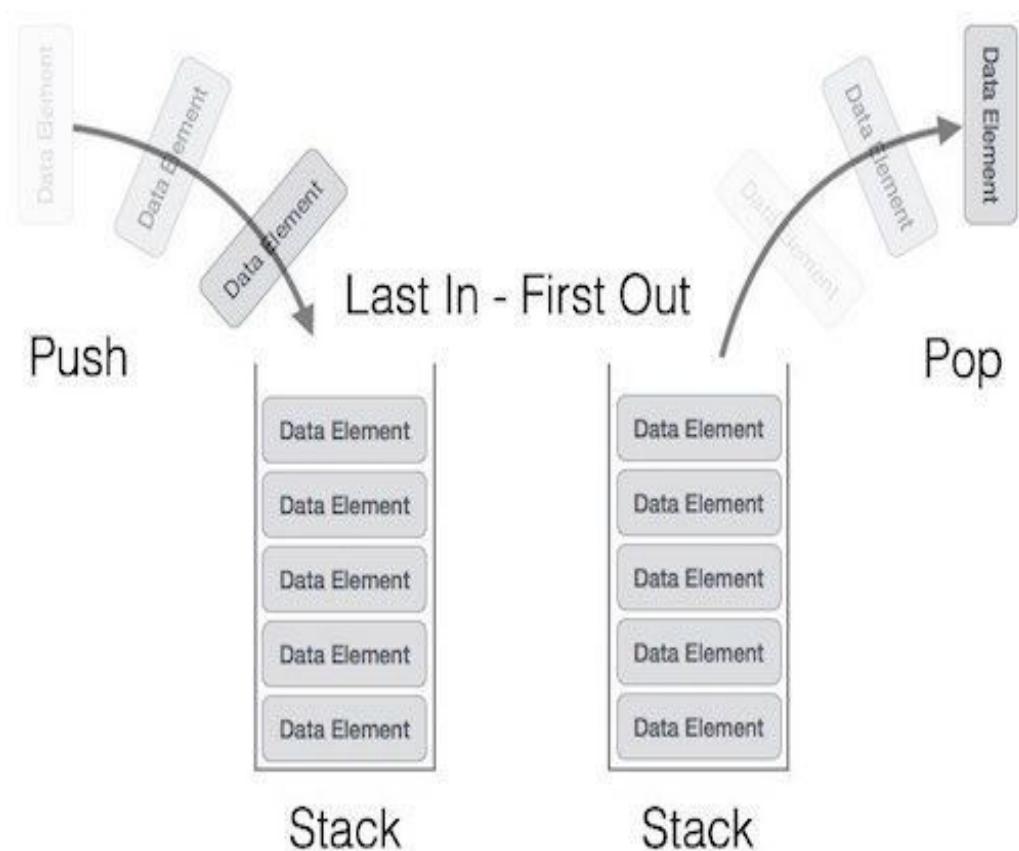
Definition: Stack is a linear data structure that principally works on the Last-in First-out (LIFO). The element that is inserted first will be out at last and vice-versa



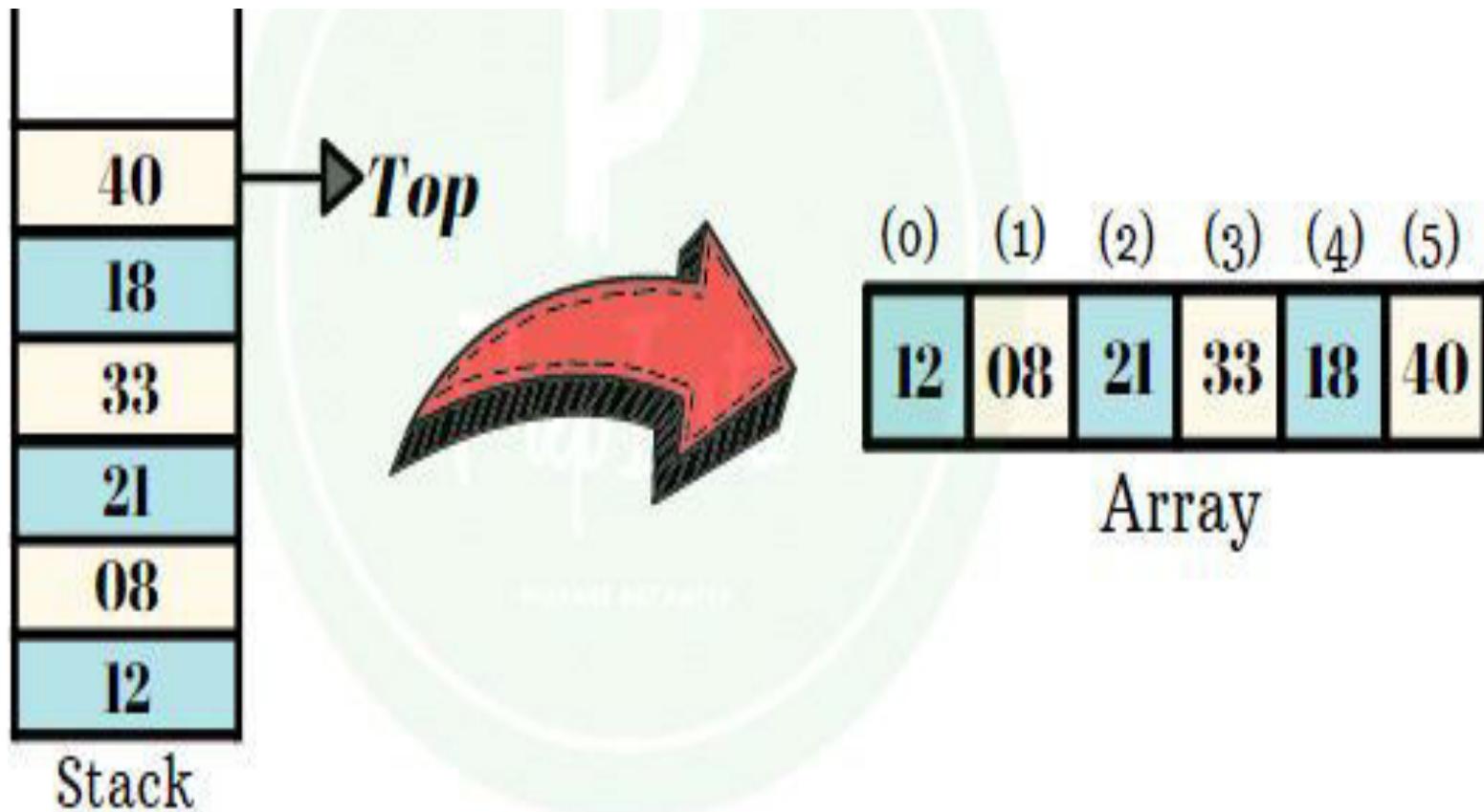
Examples of Stack

STACK TERMINOLOGIES

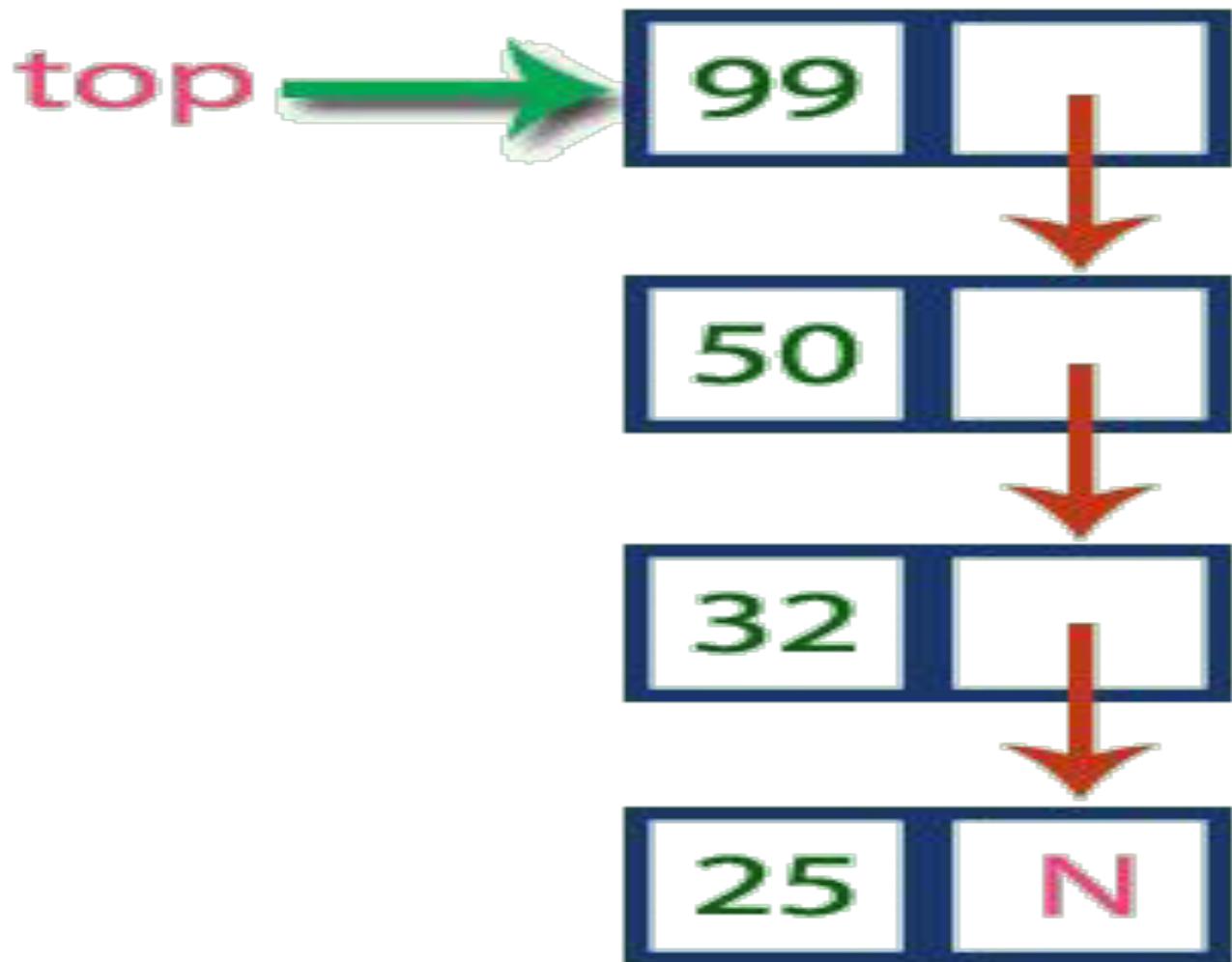
1. Top of Stack: TOP is the pointer that shows the position of the topmost element of the stack.
2. MaxStack: MaxStack shows the maximum size of the stack. ("How many elements?")
3. Stack Operations:
 - PUSH()
 - POP()



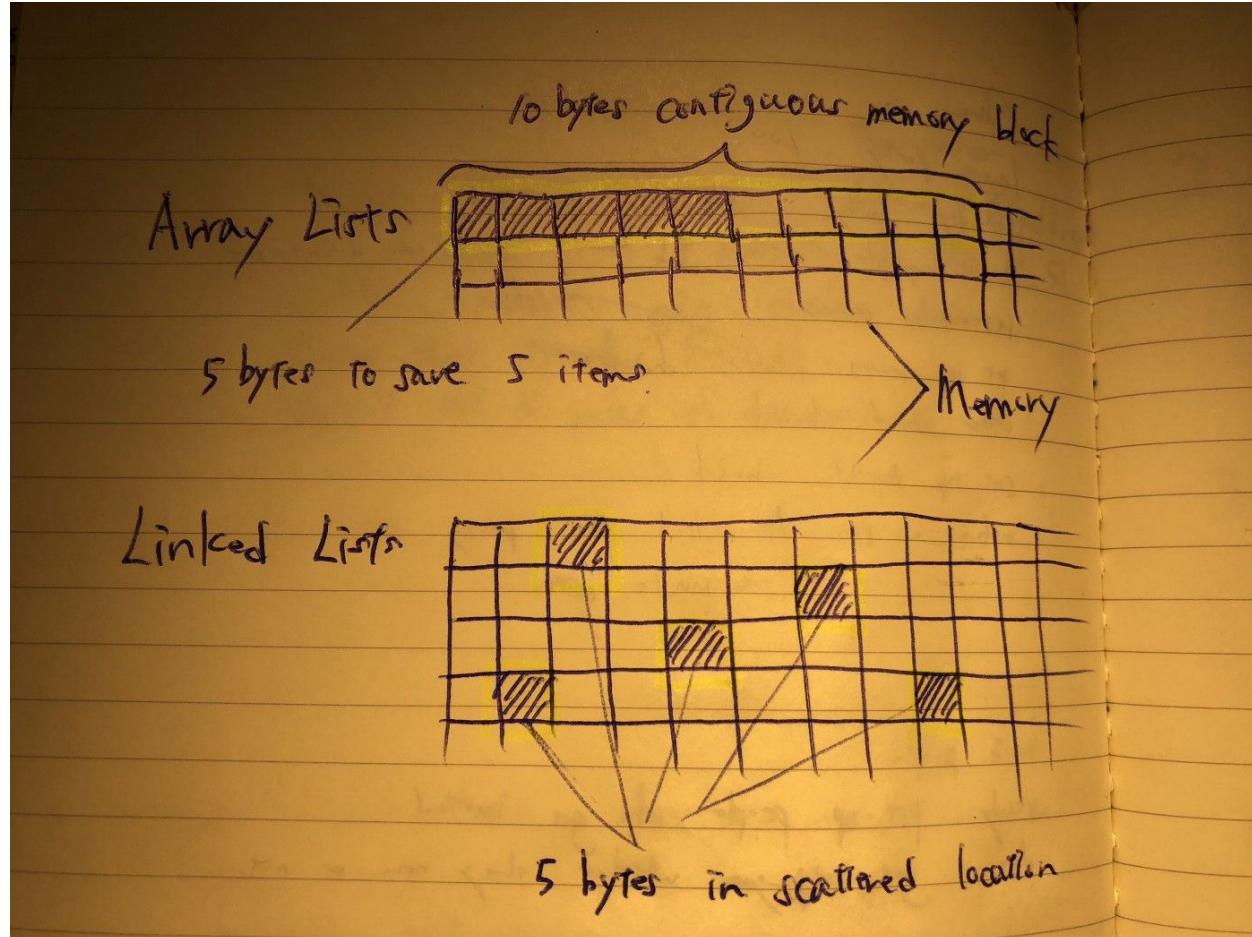
STACK: ARRAY REPRESENTATION



STACK: LIST REPRESENTATION



Memory Storage



STACK OPERATIONS

PUSH(): PUSH operation is used to push (Insert) an element onto the stack.

ALGO: PUSH()

Step 1: If $\text{TOP} \geq \text{MAXSIZE} - 1$

Then

“Stack is Overflow”

Step 2: $\text{TOP} = \text{TOP} + 1$

Step 3: $\text{STACK}[\text{TOP}] = \text{ITEM}$

POP(): POP operation is used to pop (delete) an element from the stack.

ALGO: POP()

Step 1: If $\text{TOP} = -1$

then

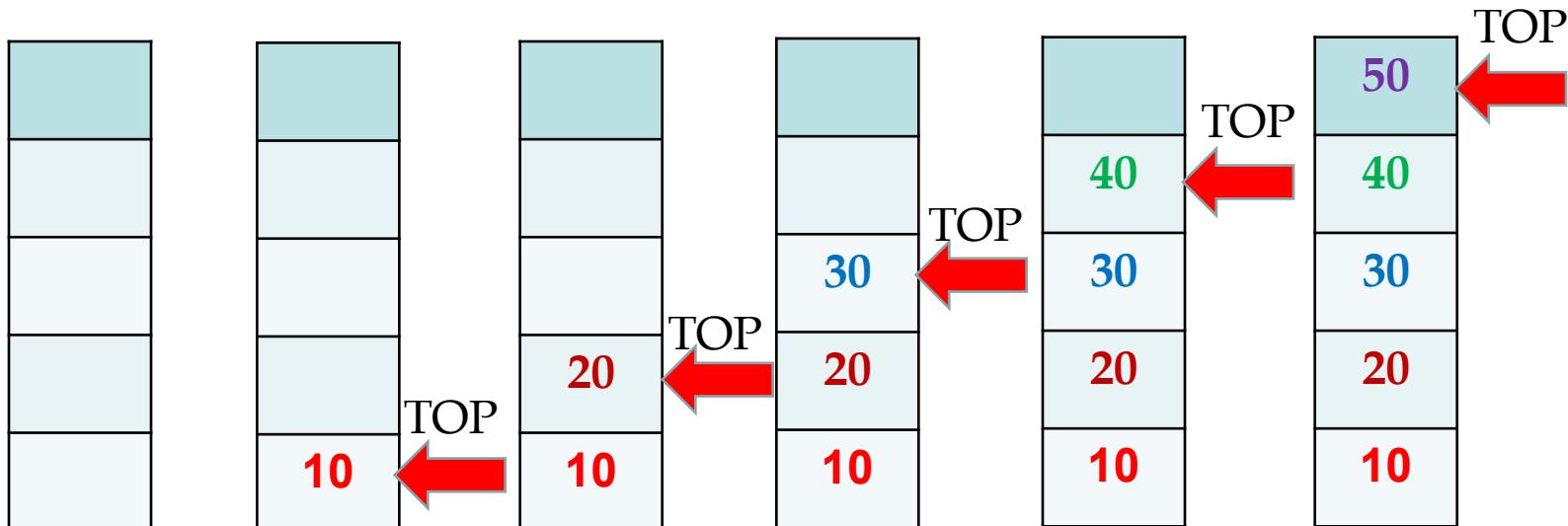
“Stack is Underflow”

Step 2: Return $\text{STACK}[\text{TOP}]$

Step 3: $\text{TOP} = \text{TOP} - 1$

STACK OPERATIONS: EXAMPLE

Let us assume a Stack of maximum size=5



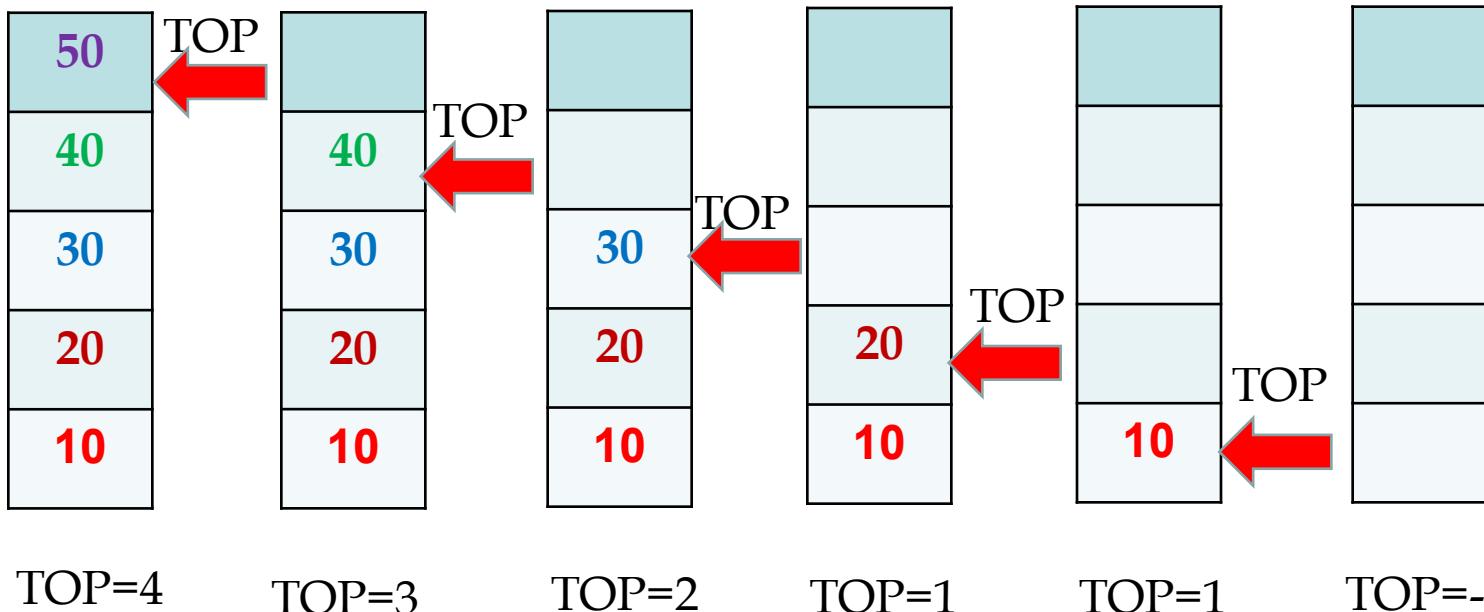
TOP=-1 TOP=0
 MaxSize=5

**PUSH
OPERATION**

When **TOP= MaxSize-1**; then stack becomes full and insertion of any new element will result in "**OVERFLOW**".

STACK OPERATIONS: EXAMPLE

Let us assume a Stack of maximum size=5



TOP=4

TOP=3

TOP=2

TOP=1

TOP=1

TOP=-1

POP OPERATION

When **TOP= -1**; then stack becomes empty and deletion of any element will result in "**UNDERFLOW**".

Time Complexity

Push() **O(1)**

Pop() **O(1)**

Display() **O(n)**



Thank You

Data Structures Using C

Course Code: CSIT124

B. Tech. (IT/CSE)

Lecture 6

Made by:

Smriti Sehgal

OBJECTIVES

- To understand basic definitions and representation of linear arrays in memory.
- To understand and implement operations on array data structure.
- To understand and implement two-dimensional arrays and their memory representation.
- To understand the use of multi-dimensional arrays and implement them.

Way-1

Formal parameters as a sized array

```
void myFunction(int param[10])  
{  
    ...  
}
```

Way-2

Formal parameters as an unsized array

```
void myFunction(int param[])  
{  
    ...  
}
```

Way-3

Formal parameters as a pointer –

```
void myFunction(int *param)  
{  
    ...  
}
```

```
#include <stdio.h>
void fun(int arr[], unsigned int n)
{
int i;
for (i=0; i<n; i++)
    printf("%d ", arr[i]);
}

// Driver program
int main()
{
int arr[] = {1, 2, 3, 4, 5, 6, 7, 8};
unsigned int n = sizeof(arr)/sizeof(arr[0]);
fun(arr, n);
return 0;
}
```

```
#include <stdio.h>
void fun(int *arr)
{
int i;
unsigned int n = sizeof(arr)/sizeof(arr[0]);
for (i=0; i<n; i++)
    printf("%d ", arr[i]);
}

// Driver program
int main()
{
int arr[] = {1, 2, 3, 4, 5, 6, 7, 8};
fun(arr);
return 0;
}
```

```
int * myFunction()
{
    ...
}
```

C programming does not allow to return an entire array as an argument

Sol:- return a pointer to an array by specifying the array's name without an index.

C does not allow to return the address of a local variable to outside of the function

Sol:- define the local variable as **static** variable.

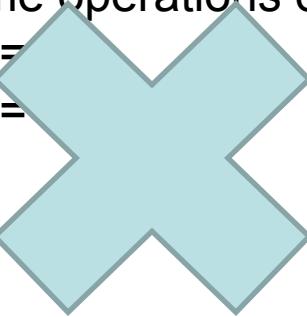
Eg

```
#include <stdio.h>
```

```
int* fun()
{
    int arr[100];

    /* Some operations on arr[] */
    arr[0] = 10;
    arr[1] = 20;

    return arr;
}
```



```
int main()
{
    int* ptr = fun();
    printf("%d %d", ptr[0], ptr[1]);
    return 0;
}
```

```
#include <stdio.h>
```

```
int* fun(int *arr)
{
    /* Some operations on arr[] */
    arr[0] = 10;
    arr[1] = 20;

    return arr;
}

int main()
{
    int arr[100];
    int* ptr = fun(arr);
    printf("%d %d", ptr[0], ptr[1]);
    return 0;
}
```

```
#include <stdio.h>

int* fun()
{
    static int arr[100];

    /* Some operations on arr[] */
    arr[0] = 10;
    arr[1] = 20;

    return arr;
}

int main()
{
    int* ptr = fun();
    printf("%d %d", ptr[0], ptr[1]);
    return 0;
}
```

```
#include <stdio.h>
```

```
int** fun()
{
    int i,j;
    int **arr;
    arr = malloc(sizeof(int*) * 2);
    for(i = 0; i < 2; i++)
    {
        arr[i] = malloc(sizeof(int*) * 2);
    }

    printf ("Enter the elements");
    for (i=0;i<2;i++)
    {
        for (j=0;j<2;j++)
            scanf("%d",&arr[i][j]);
    }

    return arr;
}
```

```
int main()
{
    int i,j;
    int** ptr = fun();
    for (i=0;i<2;i++)
    {
        for (j=0;j<2;j++)
            printf("%d",ptr[i][j]);
    }

    return 0;
}
```

Matrix in which most of the elements have **0 value**, then it is called a sparse matrix.

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix}$$

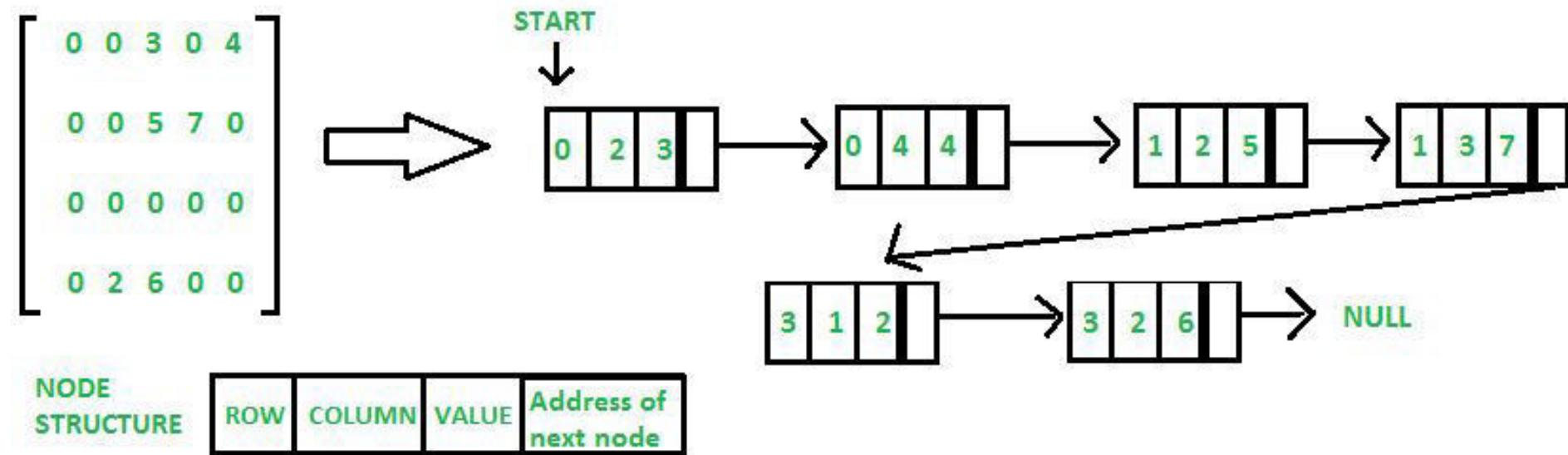
Representing a sparse matrix by a 2D array leads to wastage of lots of memory as zeroes in the matrix are of no use in most of the cases.

Sol: store non-zero elements with **triples- (Row, Column, value)**

0	0	3	0	4
0	0	5	7	0
0	0	0	0	0
0	2	6	0	0



Row	0	0	1	1	3	3
Column	2	4	2	3	1	2
Value	3	4	5	7	2	6



- Strings are defined as an array of characters.
- The difference between a character array and a string is that **the string is terminated with a special character '\0'**

Declaration of Strings
`char str_name[size];`

```
char str[] = "Amity University";  
  
char str[50] = "Amity University";  
  
char str[] = {'A','M','I','T','Y','\0'};  
  
char str[6] = {'A','M','I','T','Y','\0'};  
  
// C program to illustrate strings  
  
#include<stdio.h>  
  
int main()  
{  
    // declare and initialize string  
    char str[] = "Amity";  
  
    // print string  
    printf("%s",str);  
  
    return 0;  
}
```

```
// C program to read strings

#include<stdio.h>

int main()
{
    // declaring string
    char str[50];

    // reading string
    scanf("%s",str);

    // print string
    printf("%s",str);

    return 0;
}
```

```
// C program to illustrate how to
// pass string to functions
#include<stdio.h>

void printStr(char str[])
{
    printf("String is : %s",str);
}

int main()
{
    // declare and initialize string
    char str[] = "Amity";

    // print string by passing string
    // to a different function
    printStr(str);

    return 0;
}
```

```
#include<stdio.h>
int main()
{
    puts("Amity");
    puts("Amity%");

    getchar();
    return 0;
}
```

Amity
Amity%

```
// C program to illustrate
// gets()
#include <stdio.h>
#define MAX 15

int main()
{
    char buf[MAX];

    printf("Enter a string: ");
    gets(buf);
    printf("string is: %s\n", buf);

    return 0;
}
```

Sr.No.	Function & Purpose
1	strcpy(s1, s2); Copies string s2 into string s1.
2	strcat(s1, s2); Concatenates string s2 onto the end of string s1.
3	strlen(s1); Returns the length of string s1.
4	strcmp(s1, s2); Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	strchr(s1, ch); Returns a pointer to the first occurrence of character ch in string s1.
6	strstr(s1, s2); Returns a pointer to the first occurrence of string s2 in string s1.



Thank You

Data Structures Using C

Course Code: CSIT124

B. Tech. (IT/CSE)

Lecture #5

Made by:

Smriti Sehgal

OBJECTIVES

- To understand basic definitions and representation of linear arrays in memory.
- To understand and implement operations on array data structure.
- To understand and implement two-dimensional arrays and their memory representation.
- To understand the use of multi-dimensional arrays and implement them.

The syntax for creating a two-dimensional array is:

<data type> <array name>[size of Dim1][size of Dim2];
e.g., int a[6][4];

	0	1	2	n-1
0	a[0][0]	a[0][1]	a[0][2]	a[0][n-1]
1	a[1][0]	a[1][1]	a[1][2]	a[1][n-1]
2	a[2][0]	a[2][1]	a[2][2]	a[2][n-1]
3	a[3][0]	a[3][1]	a[3][2]	a[3][n-1]
4	a[4][0]	a[4][1]	a[4][2]	a[4][n-1]
.
.
n-1	a[n-1][0]	a[n-1][1]	a[n-1][2]	a[n-1][n-1]

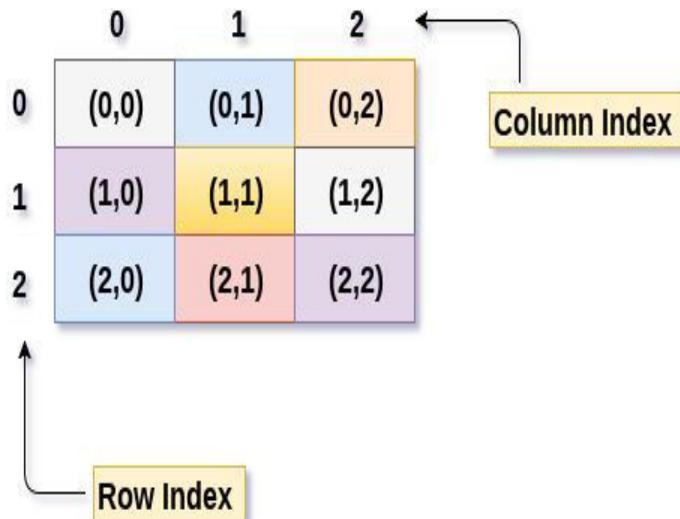
a[n][n]

```
/* declare and initialize a 2D array*/
```

```
int x[3][2] = { {19, 10},  
                {8, 17},  
                {9,15} };
```

There are two main techniques of storing 2D array elements into memory:

1. Row Major ordering



2. Column Major ordering



By Row Major Order

$$\text{Address}(a[i][j]) = \text{Base Address} + (i * n + j) * \text{size}$$

Base_Address = 2000, size= 2, n=4, m=2, i=1, j=2

$$\text{LOC } (A [i, j]) = \text{Base_Address} + \text{size} * [n * (i) + (j)]$$

$$\text{LOC } (A[1, 2]) = 2000 + 2 * [4 * (1) + 2]$$

$$= 2000 + 2 * [4 + 2]$$

$$= 2000 + 2 * 6$$

$$= 2000 + 12$$

$$= 2012$$

By Column major order

$$\text{Address}(a[i][j]) = \text{Base Address} + W * [(j * M) + i]$$

Base_Address = 2000, W= 2, N=4, M=2, i=1, j=2

$$\text{LOC } (A[i, j]) = \text{Base_Address} + W * [M * (j) + (i)]$$

$$\text{LOC } (A[1, 2]) = 2000 + 2 * [2 * (2) + 1]$$

$$= 2000 + 2 * [4 + 1]$$

$$= 2000 + 2 * 5$$

$$= 2000 + 10$$

$$= 2010$$

1D array

7	2	9	10
---	---	---	----

axis 0 →

shape: (4,)

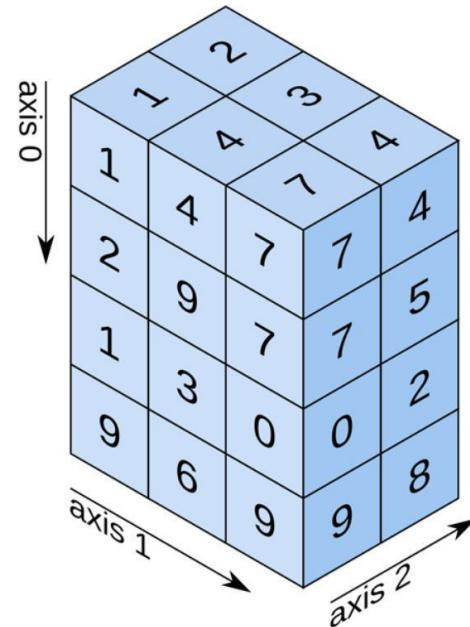
2D array

axis 0	5.2	3.0	4.5
axis 1	9.1	0.1	0.3

axis 1 →

shape: (2, 3)

3D array



shape: (4, 3, 2)

Addition of two 2d-arrays

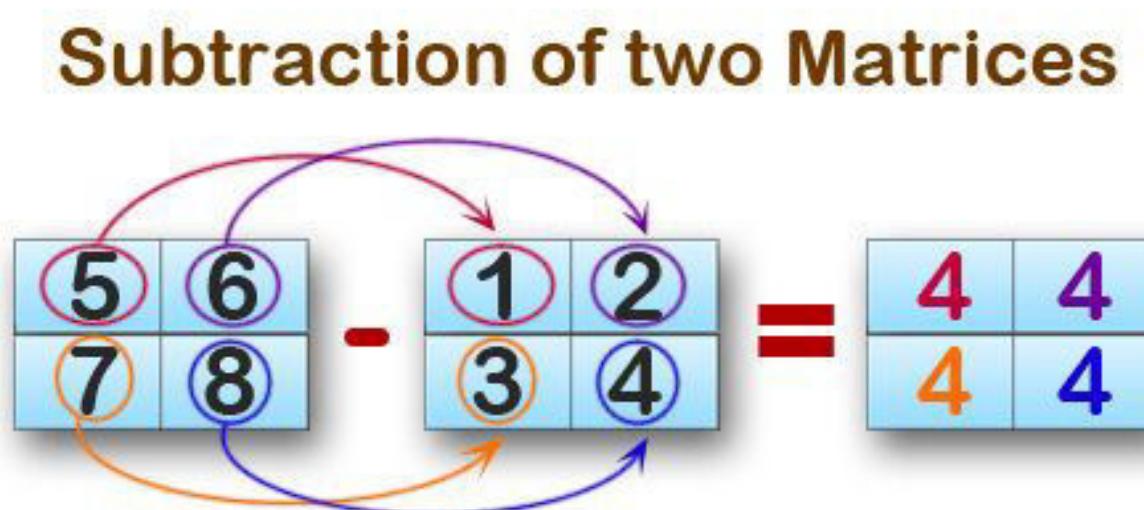
Program for addition of two matrices

$$\begin{bmatrix} 0 & 4 \\ 1 & 3 \end{bmatrix} + \begin{bmatrix} 3 & 2 \\ 5 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 0+3 & 4+2 \\ 1+5 & 3+1 \end{bmatrix} \rightarrow \begin{bmatrix} 3 & 6 \\ 6 & 4 \end{bmatrix}$$

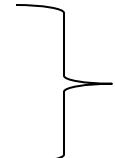
1. Input the order of the matrices.
 2. Input the matrix 1 elements.
 3. Input the matrix 2 elements.
 4. Repeat from $i = 0$ to m
 5. Repeat from $j = 0$ to n
 6. $\text{mat3}[i][j] = \text{mat1}[i][j] + \text{mat2}[i][j]$
 7. Print mat3.
- $$\left. \begin{array}{l} \\ \\ \\ \\ \\ \end{array} \right\} O(m*n)$$

Time Complexity: $O(m*n)$

Subtraction of two 2d-arrays



© w3resource.com

1. Input the order of the matrices.
 2. Input the matrix 1 elements.
 3. Input the matrix 2 elements.
 4. Repeat from $i = 0$ to m
 5. Repeat from $j = 0$ to n
 6. $\text{mat3}[i][j] = \text{mat1}[i][j] - \text{mat2}[i][j]$
 7. Print mat3.
- 
- $O(m*n)$

Time Complexity: $O(m*n)$

Multiplication of two 2d-arrays

$$\begin{bmatrix} a_{11}, a_{12} \\ a_{21}, a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11}, b_{12} \\ b_{21}, b_{22} \end{bmatrix}$$

$$\begin{bmatrix} a_{11} \times b_{11} + a_{12} \times b_{21}, & a_{11} \times b_{12} + a_{12} \times b_{22} \\ a_{21} \times b_{11} + a_{22} \times b_{21}, & a_{21} \times b_{12} + a_{22} \times b_{22} \end{bmatrix}$$

1	2
3	4

X

5	6
7	8

$$\begin{bmatrix} 1 \times 5 + 2 \times 7, & 1 \times 6 + 2 \times 8 \\ 3 \times 5 + 4 \times 7, & 3 \times 6 + 4 \times 8 \end{bmatrix} = \begin{bmatrix} 5 + 14, & 6 + 16 \\ 15 + 28, & 18 + 32 \end{bmatrix}$$

$$= \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

1. Input the order of the matrix1 (m * n).
2. Input the order of matrix2 (p * q). [n should be equal to p]
3. Input the matrix 1 elements.
4. Input the matrix 2 elements.
5. Repeat from **c** = 0 to m
6. Repeat from **c** = 0 to q
7. repeat from **k** = 0 to p
8. sum=sum+ mat1[**c**][**k**] * mat2[**k**][**d**];
9. mat3[**c**][**d**]=sum
10. Print mat3.

$O(m \cdot q \cdot p)$

Time Complexity: $O(m \cdot q \cdot p)$



Thank You

Data Structures Using C

Course Code: CSIT124

B. Tech. (IT/CSE)

Made by:

Smriti Sehgal

OBJECTIVES

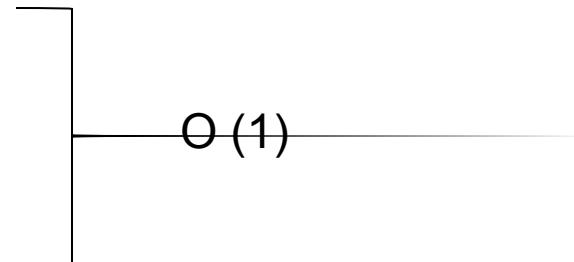
- To understand basic definitions and representation of linear arrays in memory.
- To understand and implement operations on array data structure.
- To understand and implement two-dimensional arrays and their memory representation.
- To understand the use of multi-dimensional arrays and implement them.

- **Insertion at the beginning of an array**
- Insertion at the given index of an array
- Insertion after the given index of an array
- Insertion before the given index of an array

Insertion at the Beginning of an Array

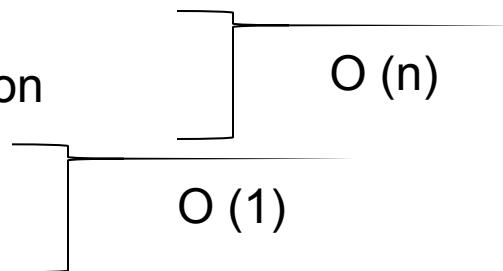
```

begin
  if N = MAX,
    return
  else
    N = N + 1
  
```

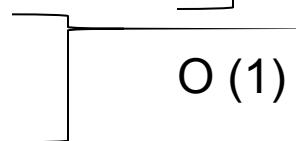


Total Time Complexity:
 $O(n)$

For All Elements in A
 Move to next adjacent location



$A[FIRST] = New_Element$

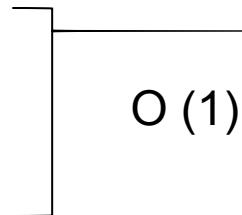


- Insertion at the beginning of an array
- **Insertion at the given index of an array**
- Insertion after the given index of an array
- Insertion before the given index of an array

Insertion at the given index of an array

begin

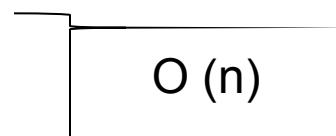
```
IF N = MAX,  
  return  
ELSE N = N + 1
```



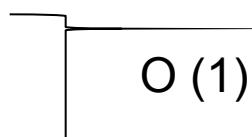
Total Time Complexity:
 $O(n)$

SEEK Location index

For All Elements from $A[\text{index}]$ to $A[N]$
 Move to next adjacent location



$A[\text{index}] = \text{New_Element}$



end

- Insertion at the beginning of an array
- Insertion at the given index of an array
- **Insertion after the given index of an array**
- Insertion before the given index of an array

Insertion after the given index of an array

begin

```
IF N = MAX,  
  return  
ELSE N = N + 1
```

O (1)

Total Time Complexity:
O(n)

SEEK Location index

For All Elements from A[index+1] to A[N]
Move to next adjacent location

O (n)

A[index+1] = New_Element

O (1)

end

- Insertion at the beginning of an array
- Insertion at the given index of an array
- Insertion after the given index of an array
- **Insertion before the given index of an array**

Insertion before the given index of an array

begin

```
IF N = MAX,  
    return  
ELSE N = N + 1
```

O (1)

Total Time Complexity:
O(n)

SEEK Location index

For All Elements from A[index-1] to A[N]
Move to next adjacent location

O (n)

A[index-1] = New_Element

O (1)

end

Program to insert a value in a 1-d array using C

Algorithm

1. Start
2. Set $J = K$
3. Repeat steps 4 and 5 while $J < N$
4. Set $LA[J] = LA[J + 1]$
5. Set $J = J+1$
6. Set $N = N-1$
7. Stop



$O(n)$

Total Time Complexity:
 $O(n)$

Deletion in an Array

Step 1:

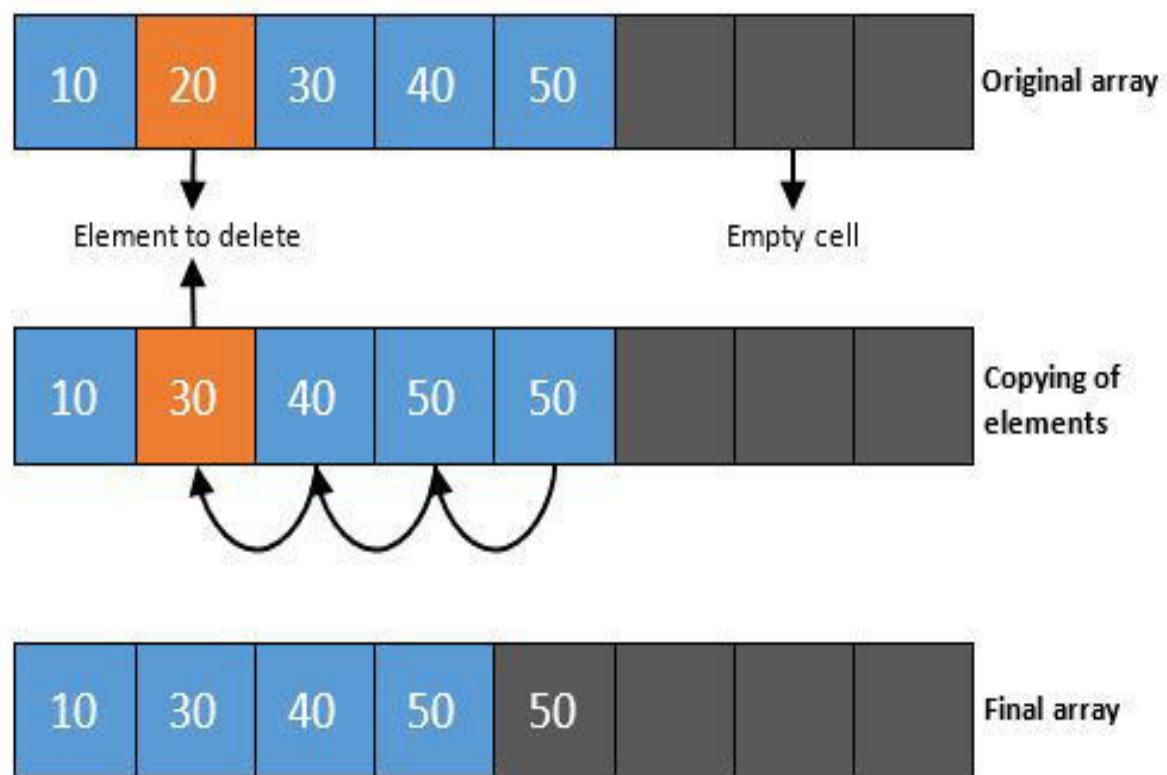
Find element to delete

Step 2:

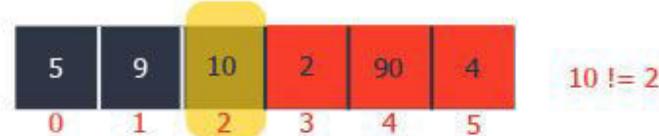
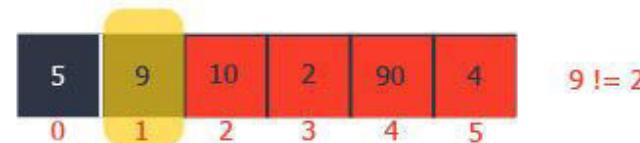
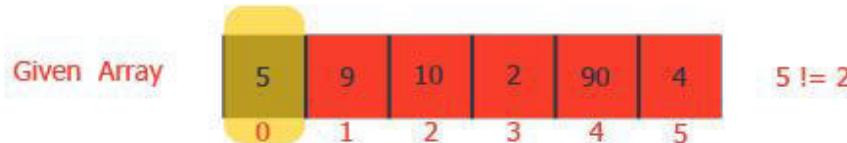
Shift elements to the left to fill the gap

Step 3:

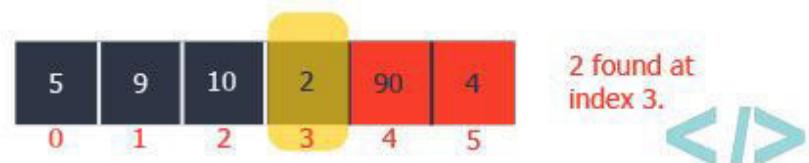
Decrease No. of elements in array by 1



Linear Search for "2" in 6 elements array



Working of Linear Search



Algorithm

Linear Search (Array A, Value x)

Step 1: Set i to 1

Step 2: if $i > n$ then go to step 7

Step 3: if $A[i] = x$ then go to step 6

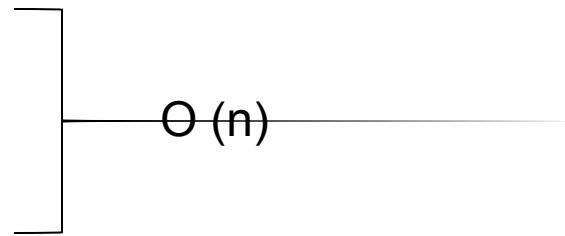
Step 4: Set i to $i + 1$

Step 5: Go to Step 2

Step 6: Print Element x Found at
index i and go to step 8

Step 7: Print element not found

Step 8: Exit



Total Time Complexity:
 $O(n)$

Binary Search

Procedure binary_search

A \leftarrow sorted array, n \leftarrow size of array, x \leftarrow value to be searched

Set lowerBound = 1, Set upperBound = n

while x not found

if upperBound < lowerBound

EXIT: x does not exists.

set midPoint = lowerBound + (upperBound - lowerBound) / 2

if A[midPoint] < x

set lowerBound = midPoint + 1

if A[midPoint] > x

set upperBound = midPoint - 1

if A[midPoint] = x

EXIT: x found at location midPoint

end while

end procedure

- At each iteration, the array is divided by half. Assume the length of an array at any iteration is n
- At **Iteration 1**, Length of array = n
- At **Iteration 2**, Length of array = $n/2$
- At **Iteration 3**, Length of array = $\frac{n/2}{2} = n/2^2$
- Therefore, after **Iteration k**, Length of array = $n/2^k$
- Also, we know that after k divisions, the **length of array becomes 1**
- Therefore, Length of array = $n/2^k = 1 \Rightarrow n = 2^k$
- Applying log function on both sides:
- $\Rightarrow \log_2(n) = \log_2(2^k)$
- $\Rightarrow \log_2(n) = k \log_2(2)$
- As $(\log_a(a) = 1)$ Therefore, $\Rightarrow k = \log_2(n)$

**Hence, the time complexity of Binary Search is
 $\log_2(n)$**

Binary Search

	0	1	2	3	4	5	6	7	8	9
Search 23	2	5	8	12	16	23	38	56	72	91
23 > 16 take 2 nd half	L=0	1	2	3	M=4	5	6	7	8	H=9
	2	5	8	12	16	23	38	56	72	91
23 > 56 take 1 st half	0	1	2	3	4	L=5	6	M=7	8	H=9
Found 23, Return 5	2	5	8	12	16	23	38	56	72	91
	0	1	2	3	4	L=5, M=5	H=6	7	8	9
	2	5	8	12	16	23	38	56	72	91



Thank You

Data Structures Using C

Course Code: CSIT124

B. Tech. (IT/CSE)

Made By:

Smriti Sehgal

OBJECTIVES

- To understand basic definitions and representation of linear arrays in memory.
- To understand and implement operations on array data structure.
- To understand and implement two-dimensional arrays and their memory representation.
- To understand the use of multi-dimensional arrays and implement them.

```
#include <stdio.h>

int main ()
{
    int var = 20; /* actual variable declaration */
    int *ip; /* pointer variable declaration */

    ip = &var; /* store address of var in pointer variable*/

    printf("Address of var variable: %x\n", &var );
    printf("Address stored in ip variable: %x\n", ip);
    printf("Value of *ip variable: %d\n", *ip );

    return 0;
}
```

*ip = points to value which is
at address stored in ip

Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20

```
int* pc, c;
```

```
c = 5;  
pc = &c;  
c = 1;
```

```
printf("%d", c);  
printf("%d", *pc);
```

```
printf("%x", pc);  
printf("%x", *pc);  
printf("%x", &pc);
```

*pc

#5678

c



1

#1234

#5678

*pc = value stored at #5678

Output :

1
1
#5678
1
#1234

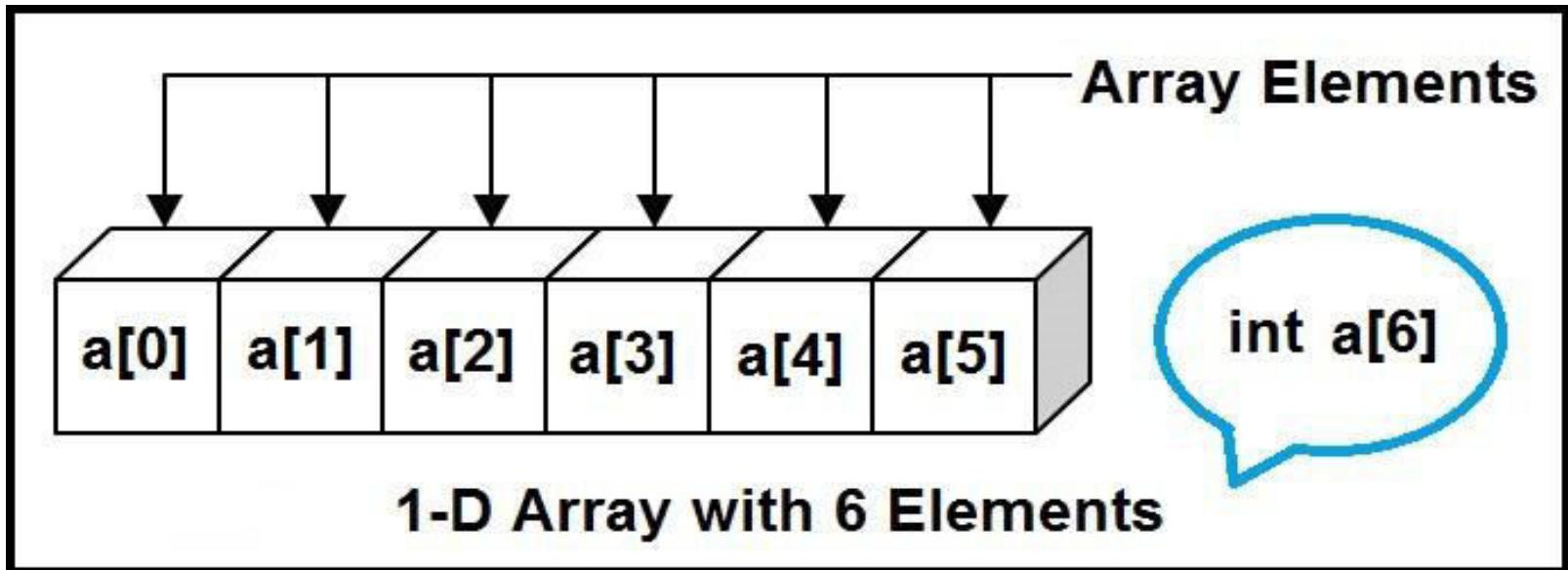
Swap using Pointers

```
#include <stdio.h>
void swap(int *x,int *y)
{
    int t;
    t = *x;
    *x = *y;
    *y = t;
}
int main()
{
    int num1,num2;
    printf("Enter value of num1: ");
    scanf("%d",&num1);
    printf("Enter value of num2: ");
    scanf("%d",&num2);

    printf("Before Swapping: num1 is: %d, num2 is: %d\n",num1,num2);
    swap(&num1,&num2);
    printf("After Swapping: num1 is: %d, num2 is: %d\n",num1,num2);
    return 0;
}
```

What is an Array?

- Array is a data structure used to store homogeneous elements at contiguous locations.
- A **one-dimensional (1 D) array** is a structured collection of components (often called array elements) that can be accessed individually by specifying the position of a component with a single index value.
- The syntax for creating a one-dimensional array:
`<data type> <array name>[size];`
e.g., `int a[6];`

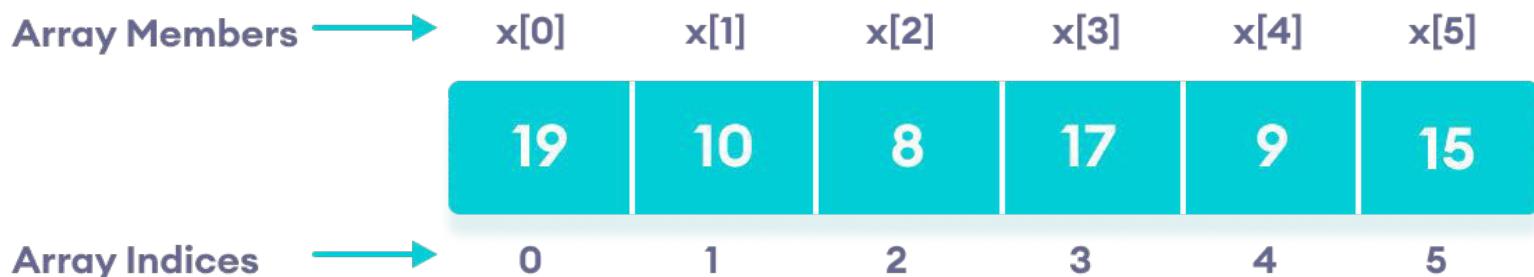


Note: Size of an array must be provided before storing data.

- In C, it's possible to initialize an array during declaration.

For example,

```
/* declare and initialize an array*/  
int x[6] = {19, 10, 8, 17, 9, 15};
```



- If an array has a size n, we can store upto n number of elements in the array.
- However, what will happen if we store less than n number of elements.
For example,

```
/*store only 3 elements in the array*/  
int x[6] = {19, 10, 8};
```

- Here, the array x has a size of 6. However, we have initialized it with only 3 elements.
- In such cases, random values (Garbage) is assigned to the remaining places.

```
#include <stdio.h>
```

```
int main()
{
    int arr[10];
    int i, N;
    printf("Enter size of array: ");
    scanf("%d", &N);
```

Static Initialization of an array

Input size of an array

```
printf("Enter %d elements in the array : ",N);
for(i=0; i<N; i++)
```

```
{  
    scanf("%d", &arr[i]);  
}
```

```
printf("\nElements in array are: ");
for(i=0; i<N; i++)
```

```
{  
    printf("%d ", arr[i]);  
}  
return 0;  
}
```

Input array elements

Display array elements

Thank You

Data Structures Using C

Course Code: CSIT124

B. Tech. (CSE)

Made by:

Ms. Smriti Sehgal

OBJECTIVES

- To understand basic definitions and representation of linear arrays in memory.
- To understand and implement operations on array data structure.
- To understand and implement two-dimensional arrays and their memory representation.
- To understand the use of multi-dimensional arrays and implement them.

- Before the steps, define the purpose of the algorithm.
- Number of steps in an algorithm should be finite.
- Use READ for input statement and PRINT OR WRITE for output statement.
- Comments are enclosed within square brackets. []
- Use EXIT or STOP to show end of an algorithm

For Selection logic, use

Single alternative

If condition, then:

[Module A]

[End of Structure]

Double Alternative
If condition, then:
[Module B]
Else:
[Module A]
[End of If structure]

Multiple Alternative
If condition(1), Then:
[Module A]
Else If condition(2), Then:
[Module B]
Else If condition(N), Then:
[Module N]
Else:
[Module M]

For Iteration Logic:

Repeat for loop:

 Repeat for K= (R to S) by T:

 [Module]

 [End of loop]

Repeat while condition:

 [Module]

 [End of loop]

Problem statement: WAP to search an element in an array using Linear Search

Algorithm

Linear_Search (Array A, Value n, Value x)

- Step 1: Initialize the variable, i.
- Step 2: repeat for $i=0$ to n
- Step 3: if $A[i] = x$,
then: return the value of i
- Step 4: increment i by 1
- Step 5: [end of for loop]
- Step 6: return -1
- Step 7: [end of function]

Corresponding C Code

```
int Linear_Search(int arr[], int n, int x)
{
    int i;
    for (i = 0; i < n; i++)
        if (arr[i] == x)
            return i;
    return -1;
}
```

Solution: Count your steps

- Time complexity estimates the time to run an algorithm.
- It's calculated by counting elementary operations.

- Asymptotic Analysis is the big idea that handles issues in analyzing algorithms.
- In Asymptotic Analysis, we evaluate the performance of an algorithm in terms of input size.
- We calculate, how the time (or space) taken by an algorithm increases with the input size.

Three cases to analyze an algorithm:

1) Worst Case

In the worst case analysis, we calculate upper bound on running time of an algorithm.

2) Average Case

In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs.

3) Best Case

In the best case analysis, we calculate lower bound on running time of an algorithm.

```
#include <stdio.h>

int search(int arr[], int n, int x)
{
    int i;
    for (i=0; i<n; i++)
    {
        if (arr[i] == x)
            return i;
    }
    return -1;
}

int main()
{
    int arr[] = {1, 10, 30, 15};
    int x = 30;
    int n = sizeof(arr);
    printf("%d is present at index %d", x, search(arr, n, x));

    getchar();
    return 0;
}
```

- In this case, the worst case happens when the element to be searched (x in the above code) is not present in the array.
- When x is not present, the search() functions compares it with all the elements of arr[] one by one.
- Therefore, the worst case time complexity of linear search would be $\Theta(n)$.

- In this case, let us assume that all cases are uniformly distributed (including the case of x not being present in array).
- So we sum all the cases and divide the sum by $(n+1)$.

$$\frac{\sum_{i=1}^{n+1} \theta(i)}{(n+1)}$$

- In this, the best case occurs when x is present at the first location.
- The number of operations in the best case is constant (not dependent on n).
- So time complexity in the best case would be $\Theta(1)$.

Examples

O(1)

```
int x = 30;
```

O(n)

```
// Here c is a positive integer constant
for (int i = 1; i <= n; i += c)
{
  // some O(1) expressions
}
for (int i = n; i > 0; i -= c)
{
  // some O(1) expressions
}
```

O(nc)

```
for (int i = 1; i <=n; i++)
{
  for (int j = 1; j <=c; j++)
  {
    // some O(1) expressions
  }
}
```

O(Log n)

```
for (int i = 1; i <=n; i *= c)
{
  // some O(1) expressions
}
```

```
#include <stdio.h>
int main()
{
    printf("Hello World");
}
```

$O(1)$

```
#include <stdio.h>
void main()
{
    int i, n = 8;
    for (i = 1; i <= n; i++) {
        printf("Hello Word !!!\n");
    }
}
```

$O(n)$

Pseudocode:

```
list_Sum(A,n)
{ //A->array and n->number of elements in the array
total =0      // cost=1 no of times=1
for i=0 to n-1 // cost=2 no of times=n+1 (+1 for the end false
condition)
sum = sum + A[i] // cost=2 no of times=n
return sum        // cost=1 no of times=1
}
```

$O(n)$

```
for (int i = 1; i <=n; i *= c)
```

```
{ // some O(1) expressions }
```

```
for (int i = n; i > 0; i /= c)
```

```
{ // some O(1) expressions }
```

**O(Log
n)**

Find Complexity of following program

```
function(int n)
{
    if (n==1)
        return;
    for (int i=1; i<=n; i++)
    {
        for (int j=1; j<=n; j++)
        {
            printf("*");
            break;
        }
    }
}
```

Time Complexity of the function: O(n)

Find Complexity of following program

```
void function(int n)
{
    int count = 0;
    for (int i=n/2; i<=n; i++)
        for (int j=1; j<=n; j = 2 * j)
            for (int k=1; k<=n; k = k * 2)
                count++;
}
```

Time Complexity of the function: $O(n \log^2 n)$

Thank You

Data Structures Using C

Course Code: CSIT124

B. Tech. (IT/CSE)

Made by:

Ms. Smriti Sehgal

OBJECTIVES

- To understand basic definitions and types of data structures.
- To understand algorithm design and complexity.
- To understand time-space tradeoff
- To understand the use of pointers in data structures.

What is Data?

- Data is a collection of raw facts and figures.
- It is produced by many sources such as: recordings and readings from a scientific experiment, generated by some sensor or hardware, census..etc.

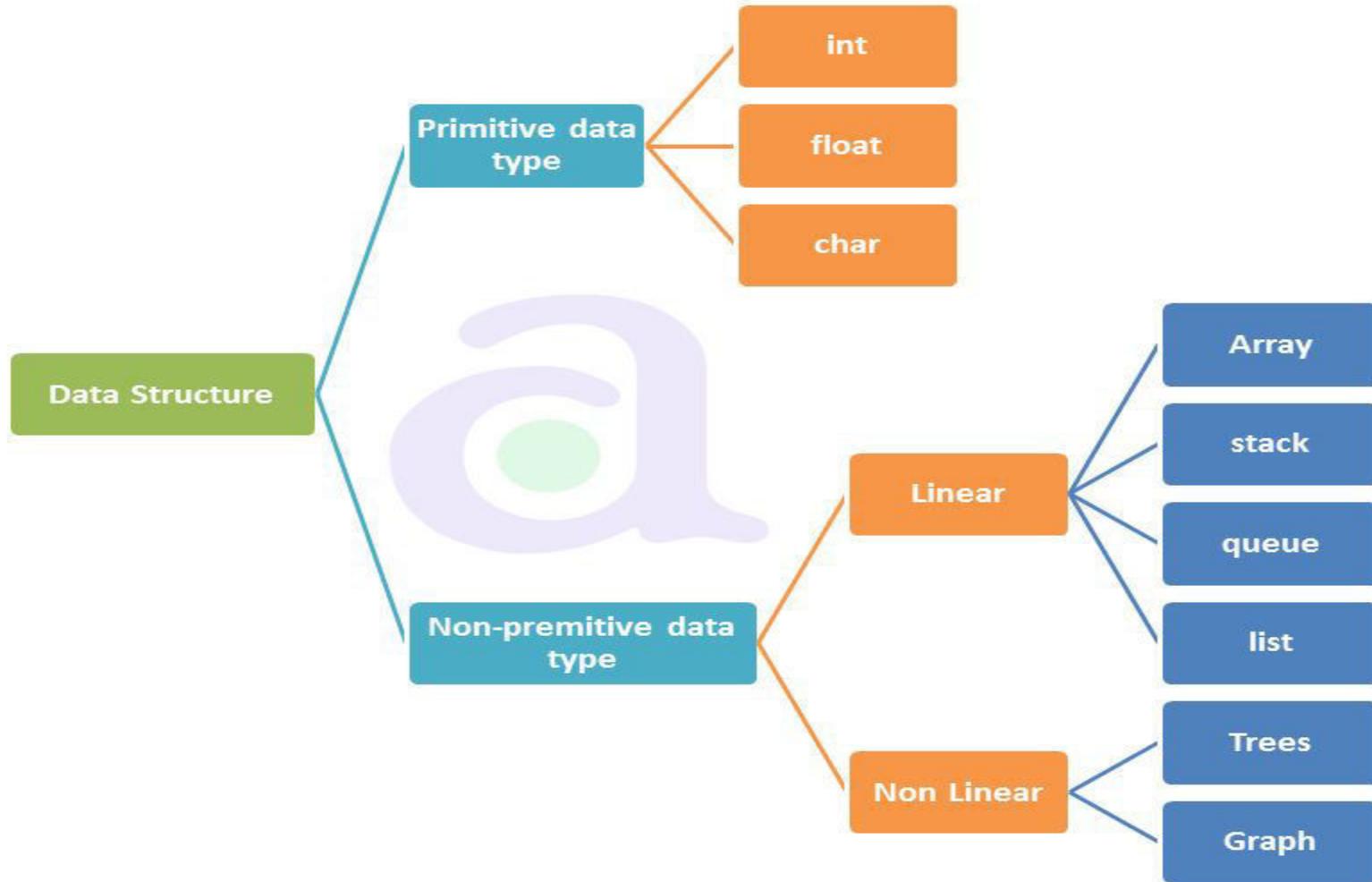


What is a Data Structure?

- A data structure is a way to organize and store this data in computer memory so that it can be managed and/or processed by the computer effectively.

- There are two major categories of data structures:
 - 1) Simple or basic
 - e.g., char, short, int, long, float, double etc.
 - 2) Complex or compound
 - A) Linear
 - e.g. arrays, stacks, queues, linked list etc.
 - B) Non-Linear
 - e.g. trees, graphs etc.

Classification



Following are the basic operations supported by a data structure:

- **Traversal** – print all the elements one by one.
- **Insertion** – add an element at given location.
- **Deletion** – delete an element at given location.
- **Search** – search an element using by value.
- **Update** – update an element at given location.

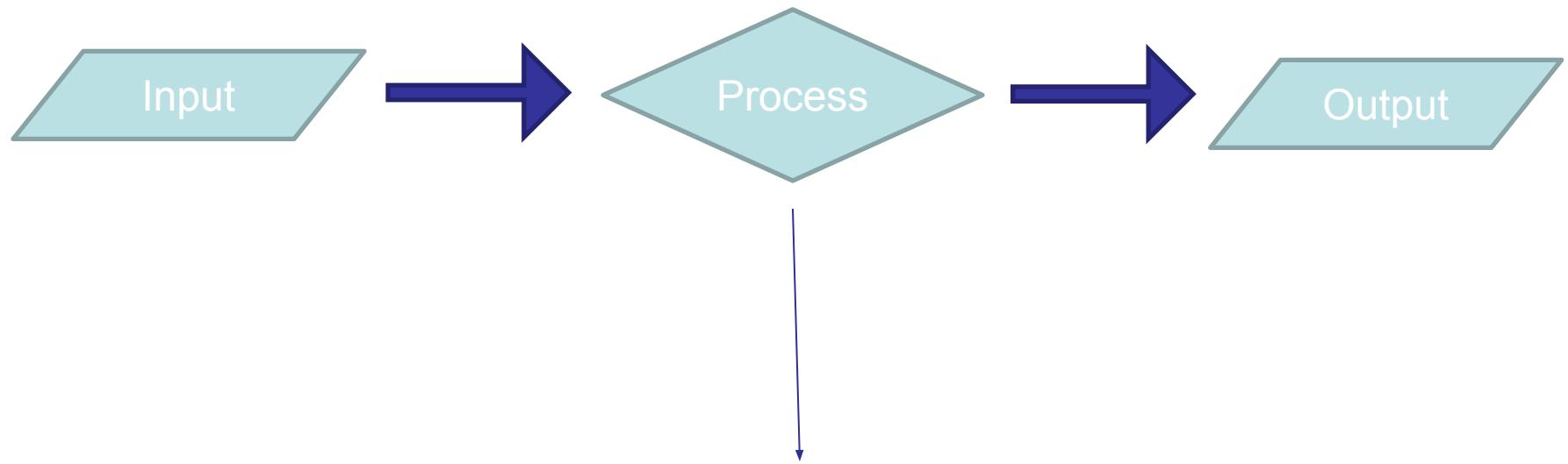
What is an algorithm?

- An algorithm is a step-by-step procedure to perform some task.
- It is the computational procedure for processing the input data and generates output i.e., processed information.

Self Reading:

- Go through some examples of algorithms and know how to design an algorithm.

Algorithm



Set of steps or instructions to process input

Linear search

Array

6	3	0	5	1	2	8	-1	4
---	---	---	---	---	---	---	----	---

Element to search: 8

[This Photo](#)This Photo by Unknown Author is licensed under [CC BY-NC-ND](#)

What do you mean by Complexity of an algorithm?

- The complexity of algorithm is a way to compare the performance of this algorithm with other similar algorithms to perform the same task.
- The complexity can be measured in terms of either “time” or “space”.
- The mathematical notations used for complexity are:
 - Ω (big Omega)
 - O (big Oh)
 - Θ (theta)

- Go through some examples of how to calculate time and space complexity of an algorithm.
- Some common examples of algorithms are:
 - 1) Traversing an array
 - 2) Searching an elements using Linear Search
 - 3) Searching an elements using Binary Search

- It states that either the time taken by the algorithm or the space occupied by it can be improved at a time.
- We can not design an algorithm which is efficient in terms of both time and space.

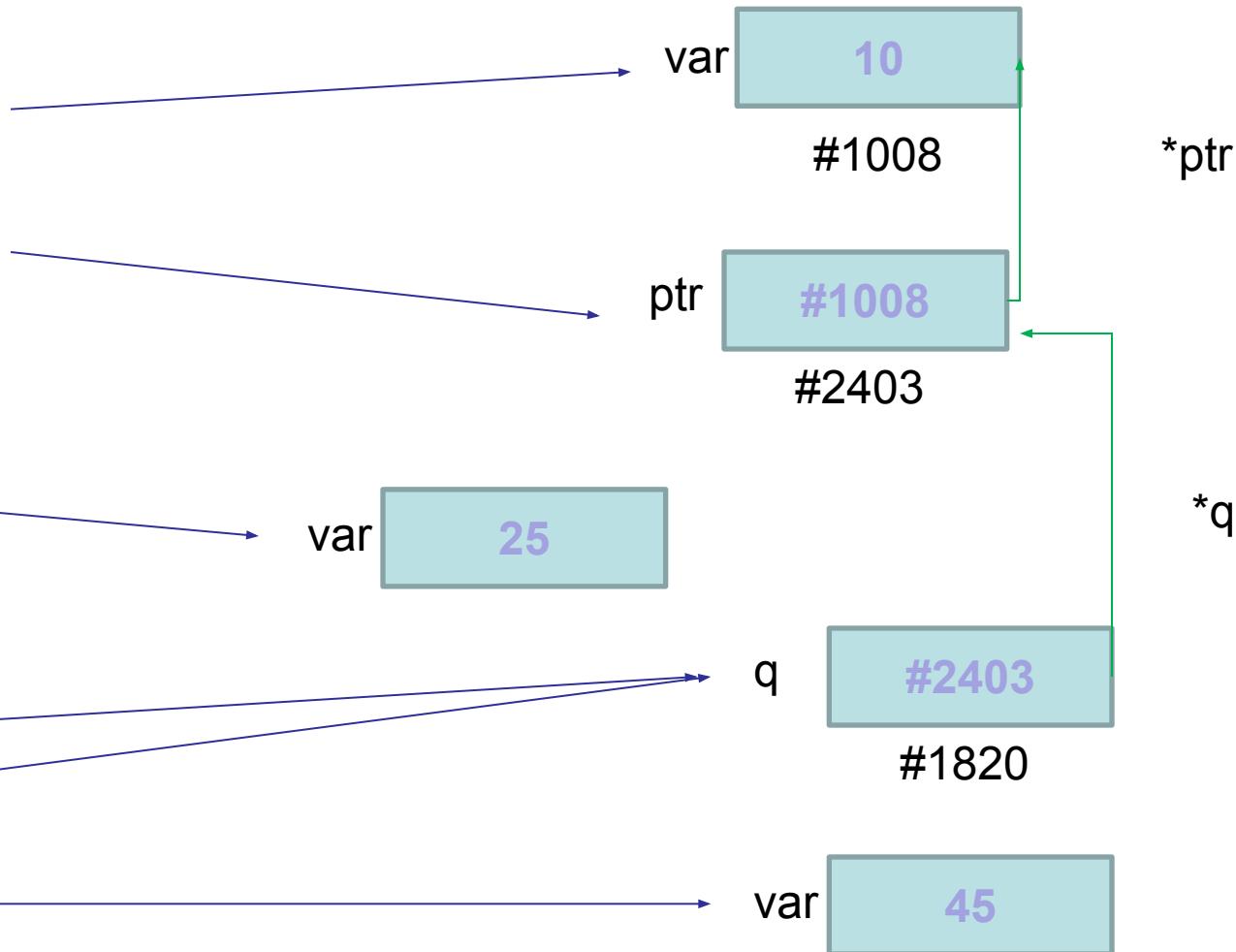
```
int var=10;
```

```
int *ptr=&var;
```

```
*ptr=25;
```

```
int **q;  
q=&ptr;
```

```
**q=45;
```



- To return more than one value from a function.
- For passing and returning arrays and strings to and from functions.
- For passing and returning different data structures to and from functions.
- For dynamic memory allocation.
- To create more complex data structures.

1. List out the areas in which data structures can be applied.

Compiler Design, Operating System, Database Management System, Statistical analysis package, Numerical Analysis, Graphics, Artificial Intelligence, Simulation

2. What are the major data structures used in the following areas :
RDBMS, Network data model & Hierarchical data model?

RDBMS - Array (i.e. Array of structures)

Network data model – Graph

Hierarchical data model - Trees

Consider the code snippet to find the sum of digits of a given number

```
#include<stdio.h>
int sum_of_digits(int n)
{
    int sm = 0;
    while(n != 0)
    { _____;
        n /= 10;
    }
    return sm;
}
int main()
{
    int n = 1234;
    int ans = sum_of_digits(n);
    printf("%d",ans);
    return 0;
}
```

Answer: sm +=
n%10

4. Which algorithmic technique does Fibonacci search use?

- a) Brute force
- b) Divide and Conquer
- c) Greedy Technique
- d) Backtracking

Answer : Divide and
Conquer

5. Which data structure is used for balancing of symbols?

- a) Stack
- b) Queue
- c) Tree
- d) Graph

Answer :
Stack

Thank you

DATA STRUCTURES USING C

Module 2 - Lecture IV

STACKS AND QUEUES

Prepared By
Ms. Neetu Narayan

Stack Objectives:

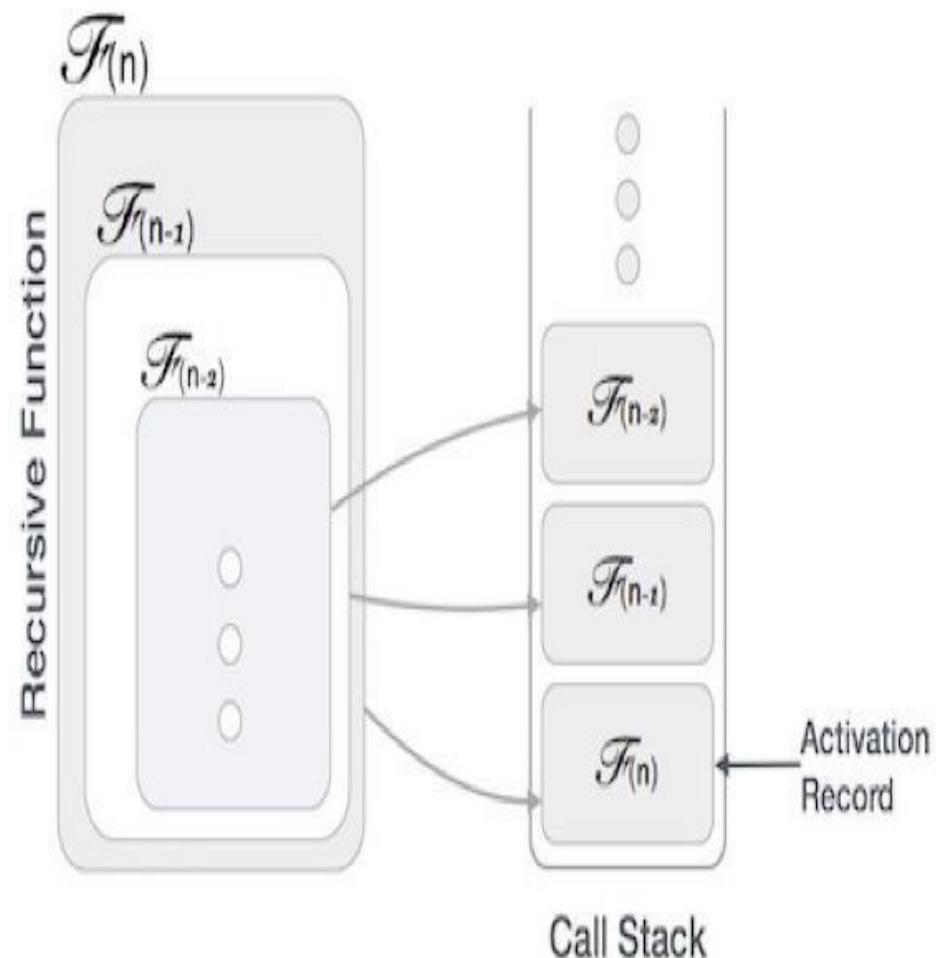
- Impart in-depth knowledge of data structure and its implementation in computer programs.
- Make students understand the concepts of Stack and Queue linear data structure.
- Illustrate asymptotic notations and their usage.

6. Recursion

Recursion is the process by which function calls itself repeatedly, until the specified condition has been satisfied.

To solve a problem recursively, two conditions must be satisfied:

1. A base value should be defined for which the function should not call itself.
2. At each function call, the argument of the function should move close to the base value.



Factorial of a number using recursion

Algorithm:

FACTO (FAC, NUMBER)

1. If NUMBER=0,
then FAC=1 and Return
2. FACTO (FAC, NUMBER-1)
3. FAC=NUMBER*FAC
4. Return

FACTO (1, 4)- Input

FACTO (1,3)

FACTO (1,2)

FACTO (1,1)

FACTO(1,0)

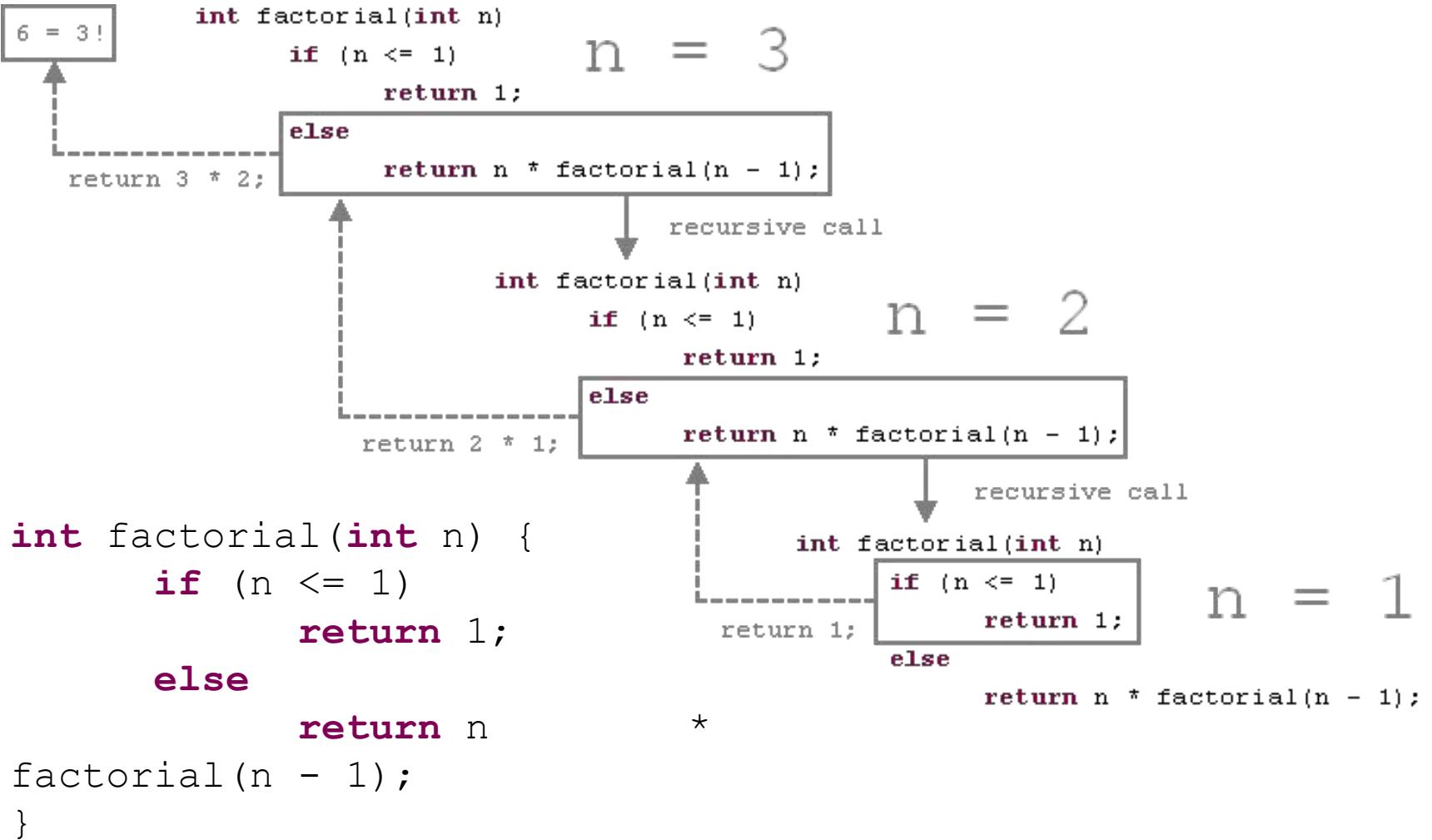
FAC=1

FACTO(1,1)=1*FACTO (1,0)=1

FACTO(1,2)=2*FACTO(1,1)=2

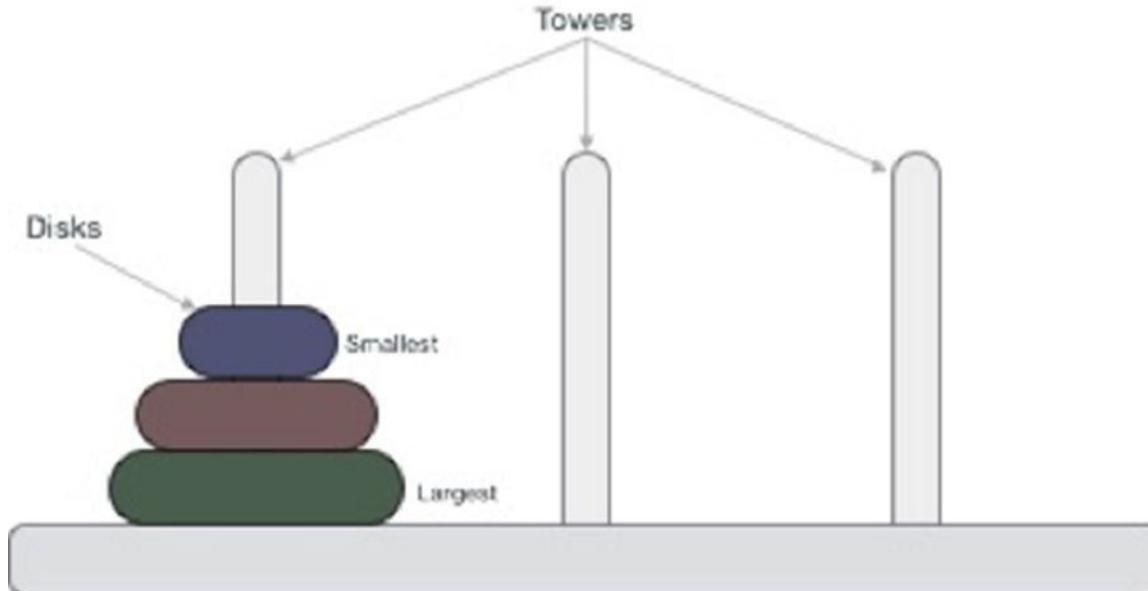
FAC= 3*FACTO(1,2)=6

FAC=4*FACTO(1,4)=24



7. Tower of Hanoi

- Mathematical puzzle
- Consists of three tower (pegs) and more than one rings;
- These rings are of different sizes and stacked upon in ascending order



- There are other variations of puzzle where the number of disks increase, but the tower count remains the same.

Rules

The mission is to move all the disks to some another tower without violating the sequence of arrangement.

1. Only one disk can be moved among the towers at any given time.
2. Only the "top" disk can be removed.
3. No large disk can sit over a small disk.

Algorithm for Towers of Hanoi

START

Procedure Hanoi(disk, source, dest, aux)

 IF disk == 1, THEN

 move disk from source to dest

 ELSE

 Hanoi(disk - 1, source, aux, dest) // Step 1

 move disk from source to dest // Step 2

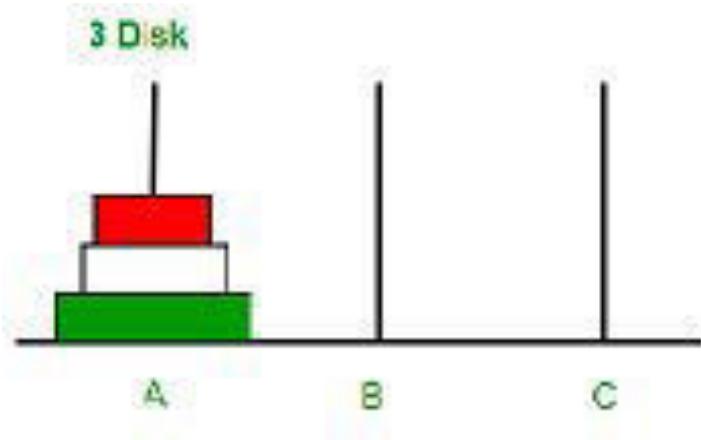
 Hanoi(disk - 1, aux, dest, source) // Step 3

 END IF

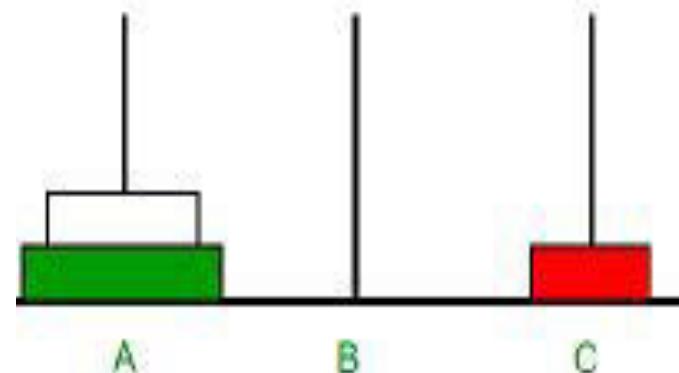
END Procedure

STOP

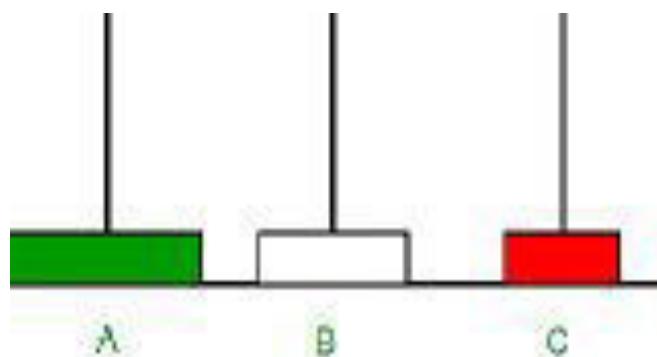
Example of Tower of Hanoi for disk=3



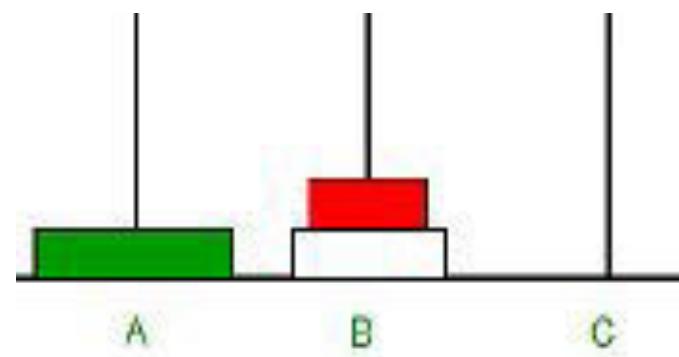
Initial



A->C

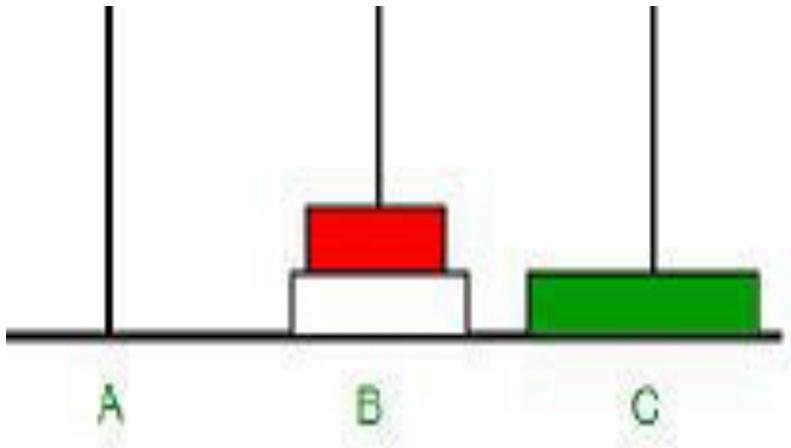


A->B

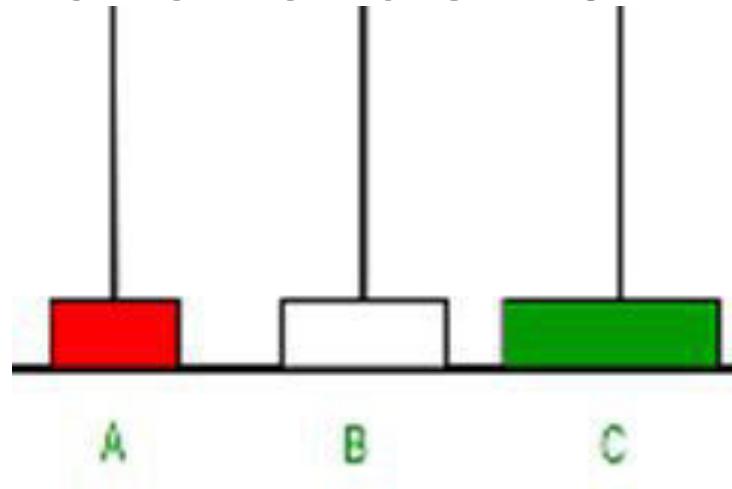


C->B

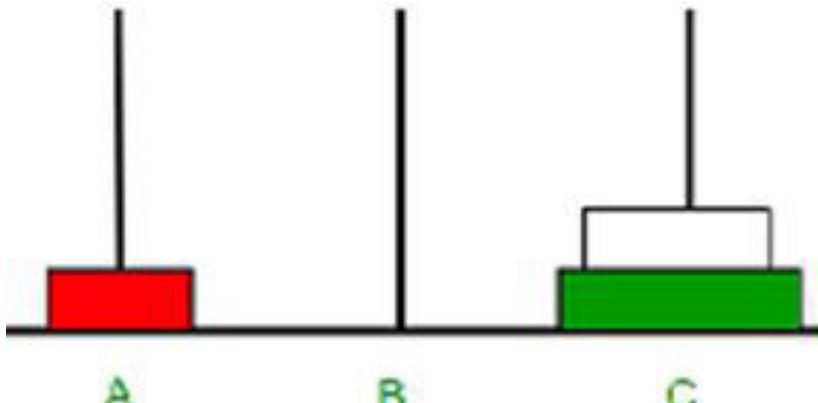
Example of Tower of Hanoi for disk=3



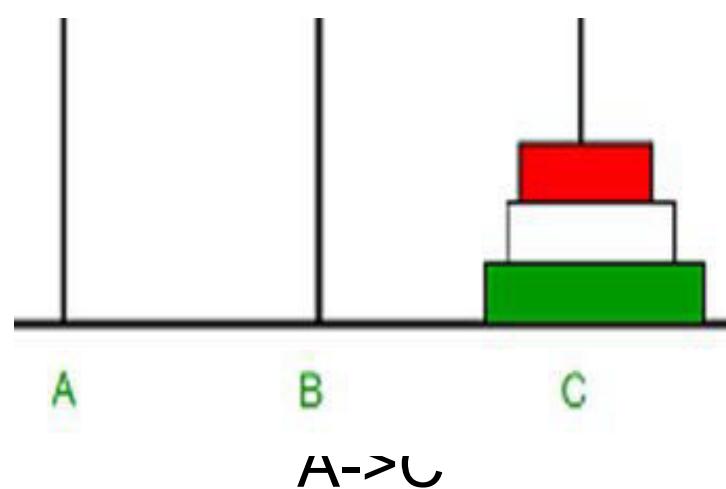
A->C



B->A



B->C



A->C

Video References

<http://towersofhanoi.info/Animate.aspx>

<http://britton.disted.camosun.bc.ca/hanoi.swf>

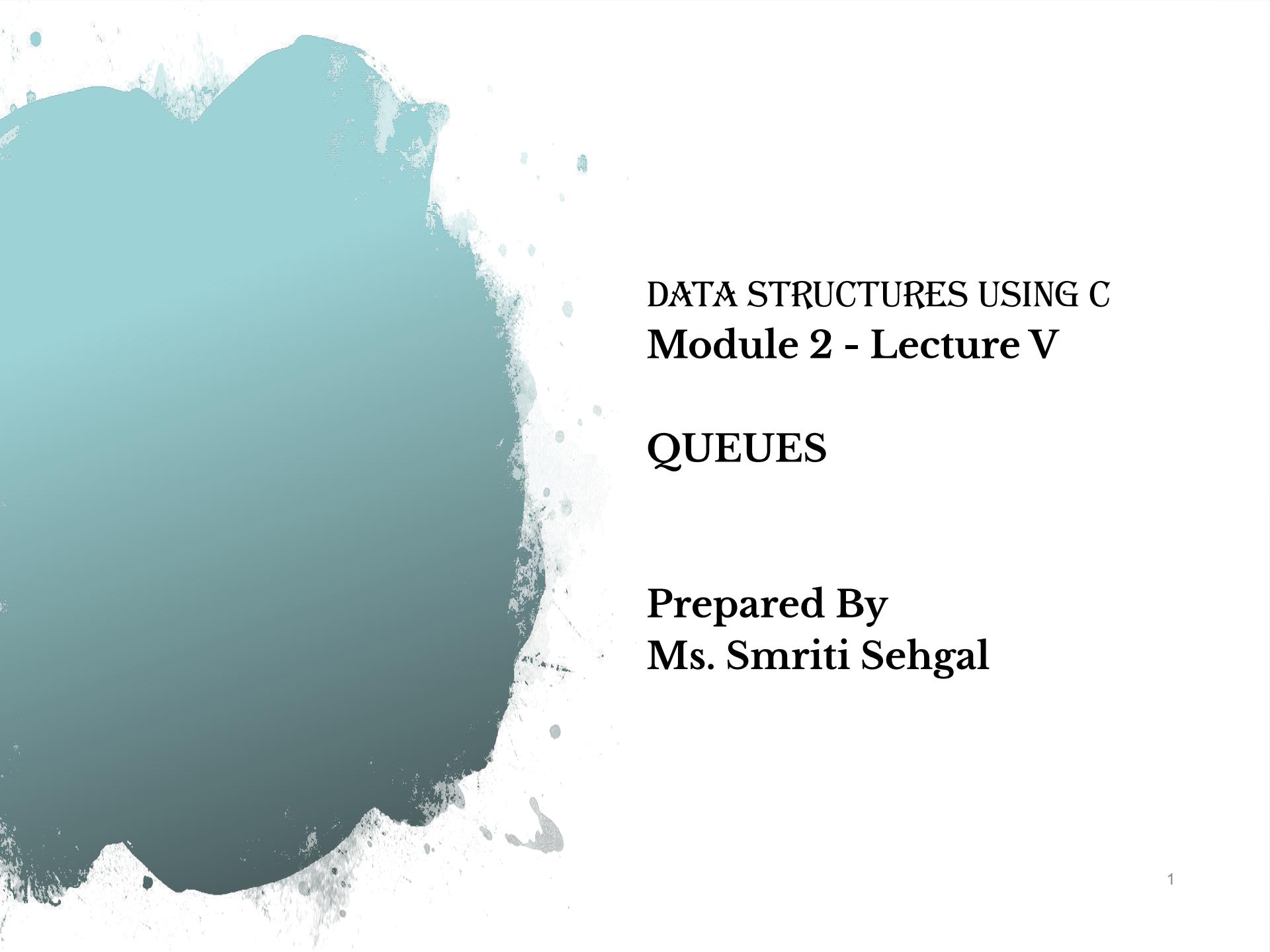
- Convert from infix to postfix & Prefix expression:
 - $((A-B)+D)/((E+F)^*G))$
 - $14/7^*3-4+9/2$
- Convert from prefix to infix expression:
 - $*-+ABCD$
 - $+-a^*BCD$

Practice Question

Amity School of Engineering & Technology

- Write a function that accepts 2 stacks.
Copy the contents of one stack to another. Note that the order of the elements must be preserved.





DATA STRUCTURES USING C

Module 2 - Lecture V

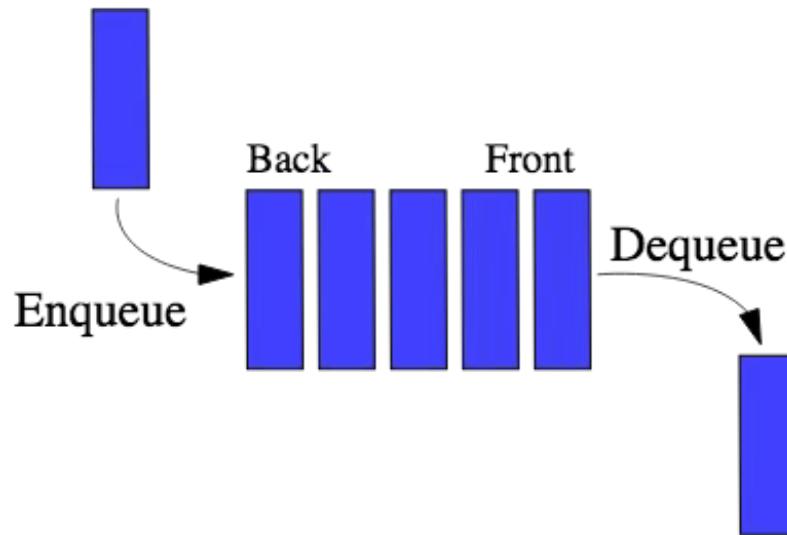
QUEUES

Prepared By
Ms. Smriti Sehgal

Queue is an abstract data structure which is open at both its ends.

One end is always used to insert data (enqueue) and the other is used to remove data (dequeue)

Queue follows First-In-First-Out methodology



Basic Operations

Queue operations involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory.

enqueue() – add (store) an item to the queue.

dequeue() – remove (access) an item from the queue.

peek() – Gets the element at the front of the queue without removing it.

isfull() – Checks if the queue is full.

isempty() – Checks if the queue is empty.

This function helps to see the data at the **front** of the queue.

Algorithm

```
begin procedure peek
    return queue[front]
end procedure
```

Example

```
int peek()
{
    return queue[front];
}
```

Algorithm

```
begin procedure isfull
    if rear equals to MAXSIZE
        return true
    else
        return false
    endif
end procedure
```

Example

```
bool isfull()
{
    if(rear == MAXSIZE - 1)
        return true;
    else
        return false;
}
```

Algorithm

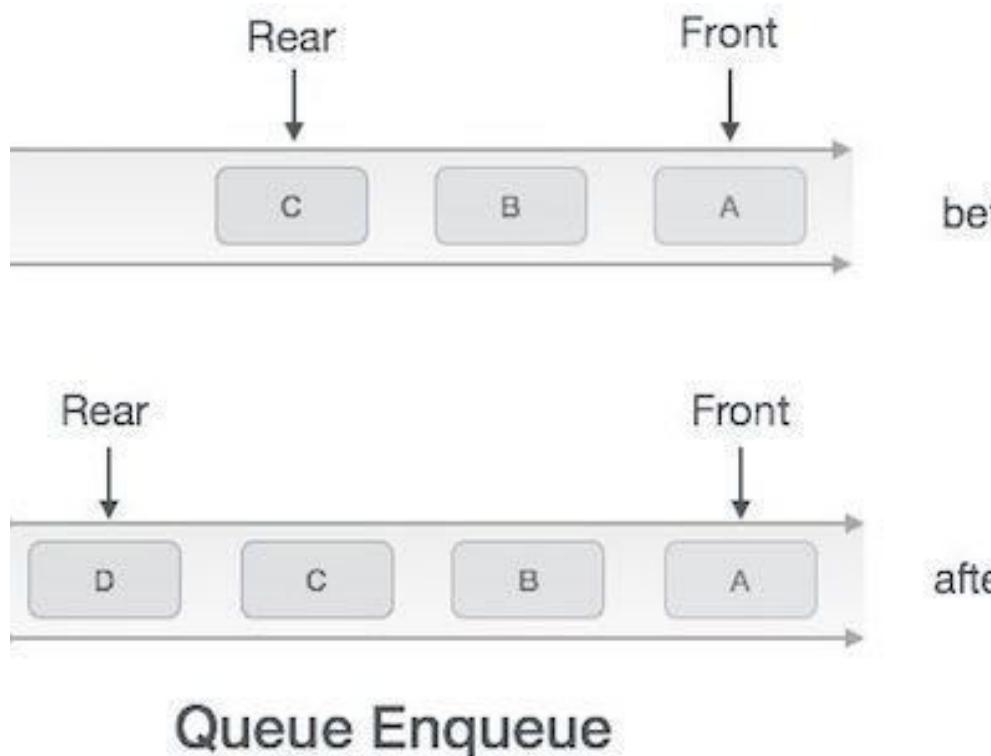
```
begin procedure isempty
  if front is less than MIN OR front is greater than rear
    return true
  else
    return false
  endif
end procedure
```

If the value of **front** is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

Example

```
bool isempty()
{
  if(front < 0 || front > rear)
    return true;
  else
    return false;
}
```

Enqueue Operation



Queues maintain two data pointers, **front** and **rear**.

- Step 1** – Check if the queue is full.
- Step 2** – If the queue is full, produce overflow error and exit.
- Step 3** – If the queue is not full, increment **rear** pointer to point the next empty space.
- Step 4** – Add data element to the queue location, where the **rear** is pointing.
- Step 5** – return success.

Accessing data from the queue is a process of two tasks

- access the data where **front** is pointing
- remove the data after access

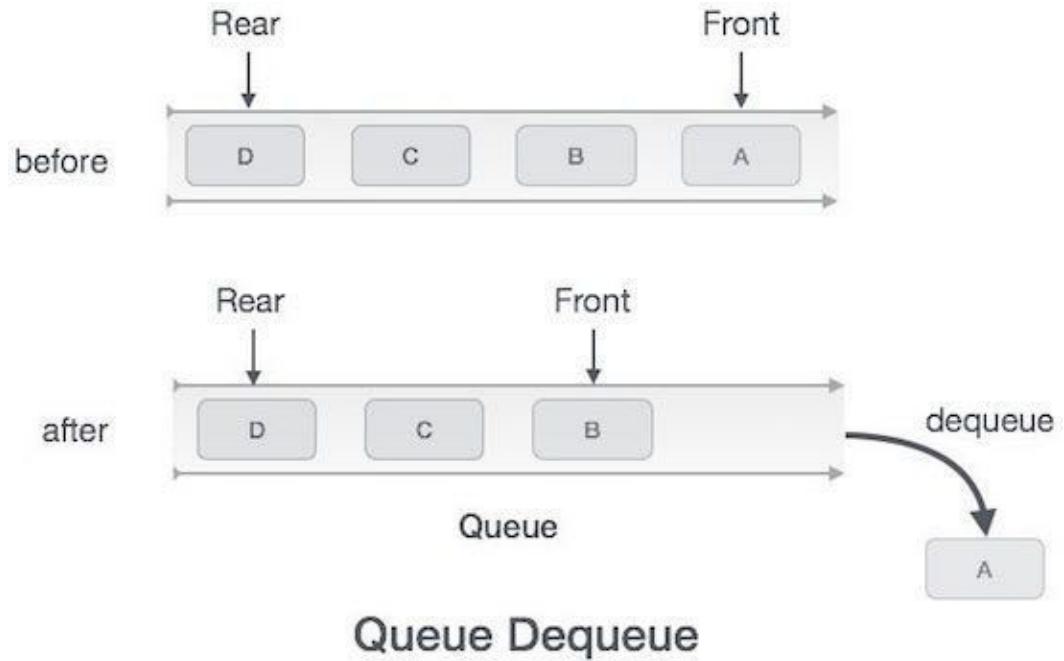
Step 1 – Check if the queue is empty.

Step 2 – If the queue is empty, produce underflow error and exit.

Step 3 – If the queue is not empty, access the data where **front** is pointing.

Step 4 – Increment **front** pointer to point to the next available data element.

Step 5 – Return success.



H	E	L	L	O	
0	1	2	3	4	5

↑
front
0

↑
rear
4

Queue

H	E	L	L	O	G
0	1	2	3	4	5

↑
front
0

↑
rear
5

Queue after inserting an element

	E	L	L	O	G
0	1	2	3	4	5

↑
front
1

↑
rear
5

Queue after deleting an element

- Add A,B,C,D,E,F =
- DELETE 2 ALPHABETS =
- ADD G =
- ADD H =
- DELETE 4 LETTERS =
- ADD I =

A,B,C,D,E,F
C,D,E,F
C,D,E,F,G
C,D,E,F,G,H
G,H
G,H,I

- FRONT=1, REAR = 5



- ADD F



- DELETE 2 ALPHABETS



- ADD G,H



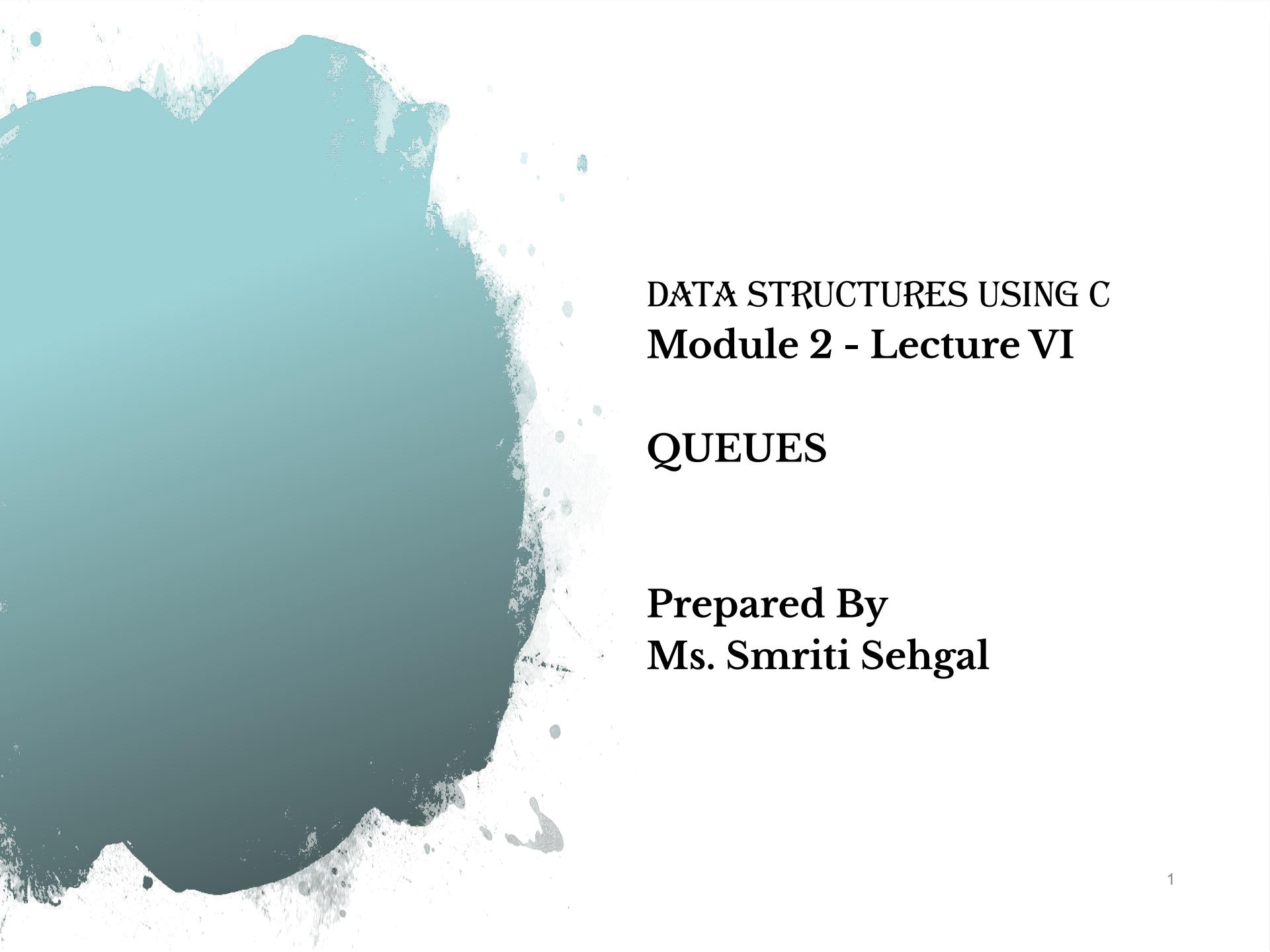
- DELETE 4 ALPHABETS



- ADD I







DATA STRUCTURES USING C

Module 2 - Lecture VI

QUEUES

Prepared By
Ms. Smriti Sehgal

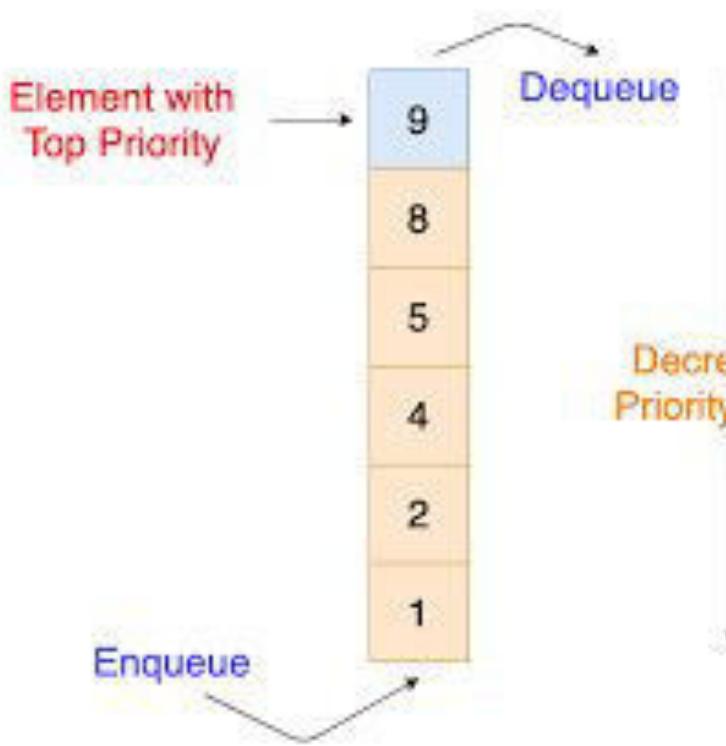
Priority Queue

- In Priority queue items are ordered by key value so that item with the lowest value of key is at front and item with the highest value of key is at rear.

- Priority is assigned to item based on its key value.

- Lower the value, higher the priority.

- A priority queue is a collection in which items can be added at any time, but the only item that can be removed is the one with the **highest priority**.



Implementation of Priority Queue

Priority queue supports following operations.

Time Complexity:
 $O(1)$ As insertion is done at last position

insert(item, priority): Inserts an item at the end with given priority.

Time Complexity: $O(n)$ As, scan through max of n elements are required.

getHighestPriority()/getLowestPriority(): Returns the highest/lowest priority item.

deleteHighestPriority()/deleteLowestPriority(): Removes the highest/lowest priority item.

Time Complexity: $O(n)$ As, scan through max of n elements are required and deletion requires shifting of all elements to left by one.

Types of Priority Queue

- Ascending (Min) priority Queue
 - Insertion can be done at any time.
 - Item with lowest priority is deleted.
- Descending (Max) priority Queue
 - Insertion can be done at any time.
 - Item with highest priority is deleted.

Working of Max-Priority Queue using Arrays

Item	Priority
10	4(lowest priority)
20	2
30	3
40	1(highest priority)

Add element 50 with priority 7

Item	Priority
10	4
20	2
30	3
40	1(highest priority)
50	7(lowest priority)

Delete

Item	Priority
10	4
20	2(highest priority)
30	3
50	7(lowest priority)

Double ended Queue (Dequeue)

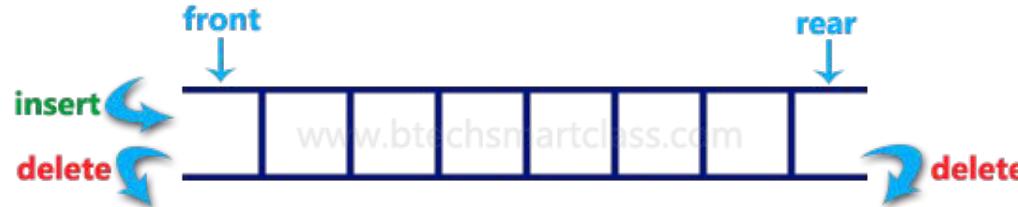
Double Ended Queue is also a Queue data structure in which the **insertion and deletion** operations are performed **at both the ends (front and rear)**.



Dequeue represented in TWO ways,

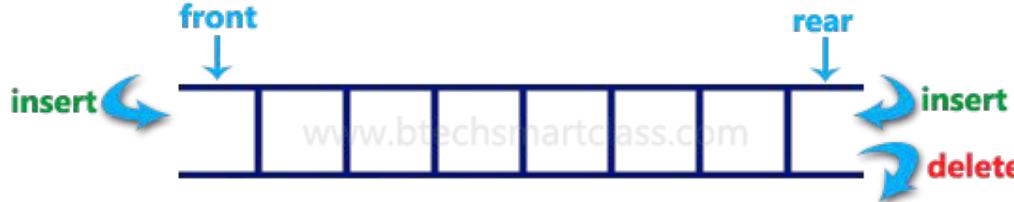
- Input Restricted Double Ended Queue
- Output Restricted Double Ended Queue

- **Input Restricted Double Ended Queue**



the insertion operation is performed at only one end and
deletion operation is performed at both the ends.

- **Output Restricted Double Ended Queue**



the deletion operation is performed at only one end.

```
int queue[SIZE];  
  
int rear = 0, front = 0;  
  
void enqueue(int);  
  
int dequeueFront();  
  
int dequeueRear();  
  
void enqueueRear(int);  
  
void enqueueFront(int);  
  
void display();
```

```
void enQueueFront(int value)  
{  
    char ch;  
    if(front==SIZE/2)  
    {   printf("\nQueue is full");  
        return;  
    }  
    do  
    {   printf("\nEnter the value");  
        scanf("%d",&value);  
        rear--;  
        queue[rear] = value;  
        ch = getch();  
    } while(ch == 'y');  
}
```

```
int deQueueRear()  
{  
    int deleted;  
    if(front == rear)  
    {  
        printf("\nQueue is Empty!");  
        return 0;  
    }  
    front--;  
    deleted = queue[front+1];  
    return deleted;  
}
```

```

void display()
{
    int i;
    if(front == rear)
        printf("\nQueue is Empty!!")
    else{
        printf("\nThe Queue elements are:");
        for(i=rear; i < front; i++)
        {
            printf("%d\t",queue[i]);
        }
    }
}

```

```

void enQueueRear(int value)
{
    char ch;
    if(front == SIZE/2)
    {   printf("\nQueue is full! ");
        return;
    }
    do {
        printf("\nEnter the value");
        scanf("%d",&value);
        queue[front] = value;
        front++;
        ch=getch();
    }while(ch=='y');
}

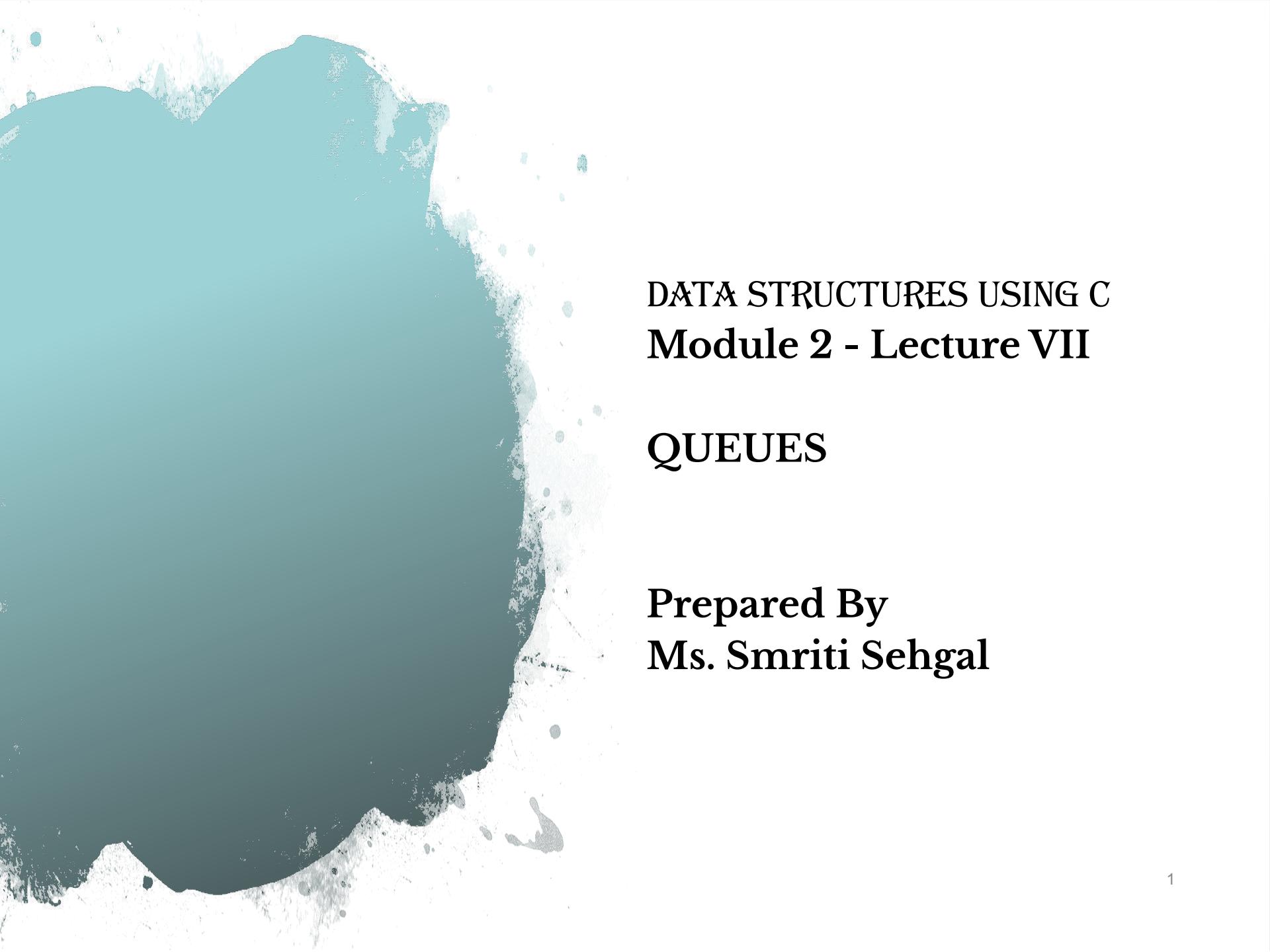
```

```

int deQueueFront()
{
    int deleted;
    if(front == rear)
    {
        printf("Queue is Empty!!!");
        return 0;
    }
    rear++;
    deleted = queue[rear-1];
    return deleted;
}

```





DATA STRUCTURES USING C

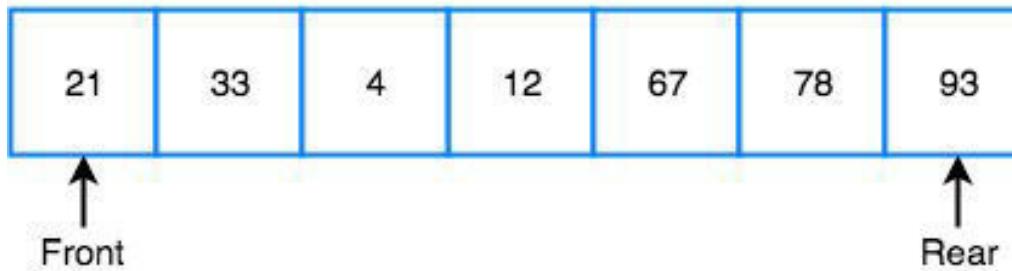
Module 2 - Lecture VII

QUEUES

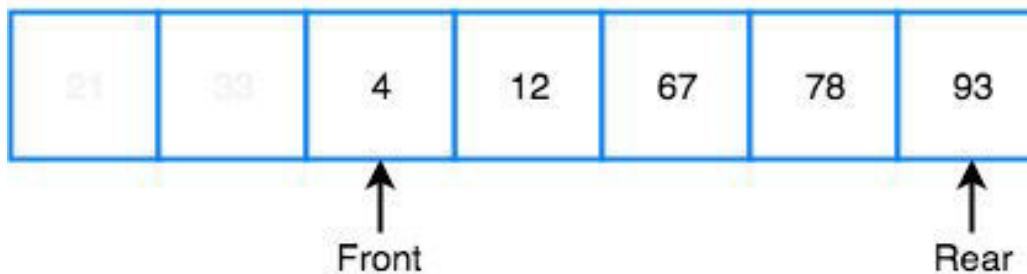
Prepared By
Ms. Smriti Sehgal

Need for Circular Queue?

Queue is Full



Queue is Full (Even after removing 2 elements)

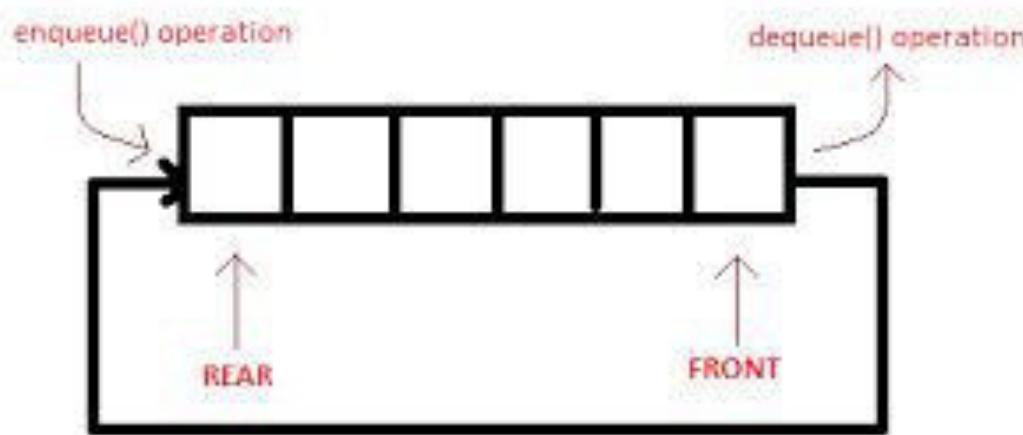


2 solutions

- Shift all elements to left and vacant spaces are occupied. Time Consuming
- Use Circular Queue

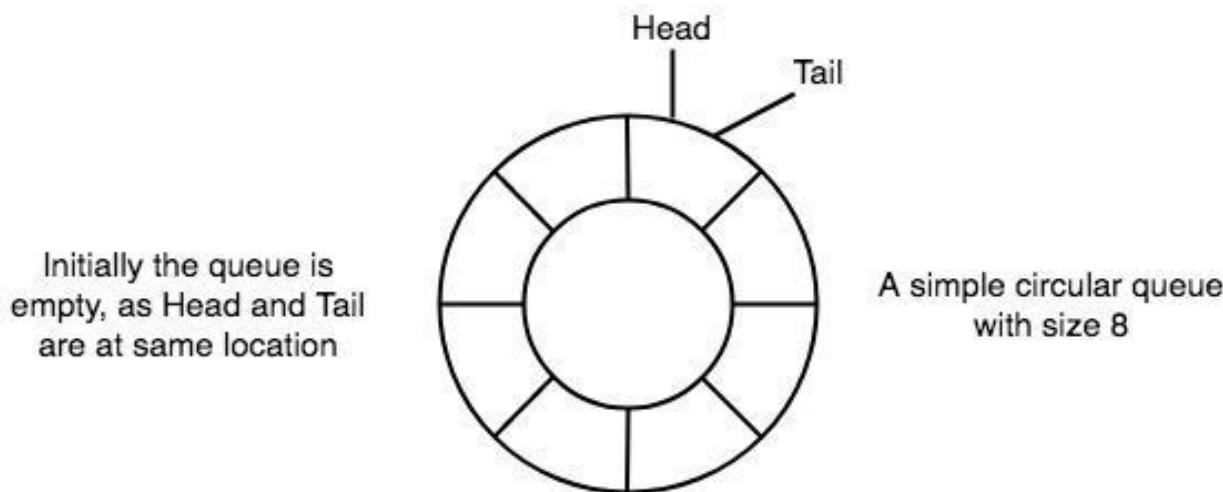
Circular Queue

Circular Queue is also a linear data structure, which follows the principle of **FIFO**(First In First Out), but instead of ending the queue at the last position, it again starts from the first position after the last, hence making the queue behave like a circular data structure.

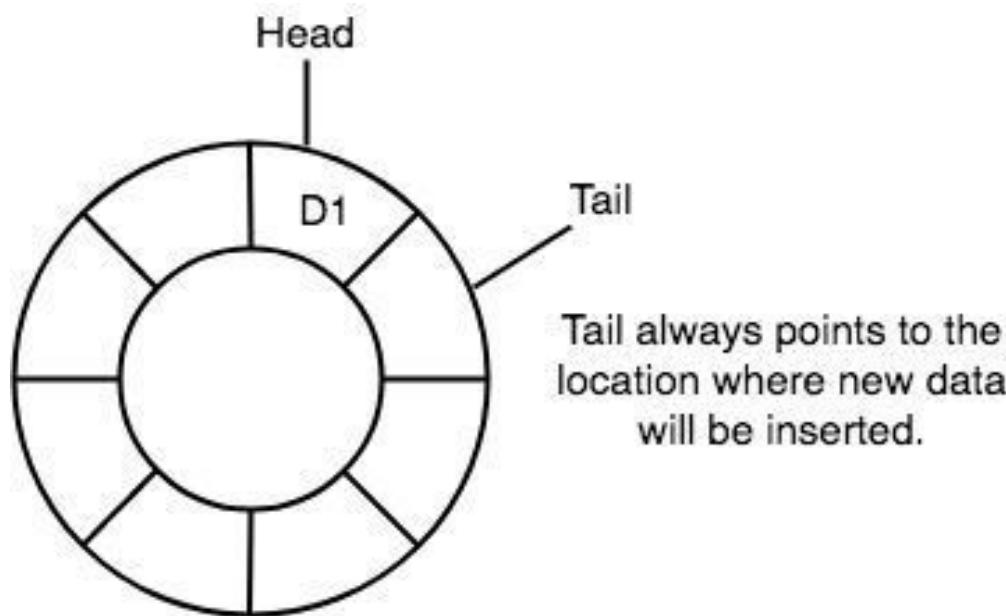


Points to be noted

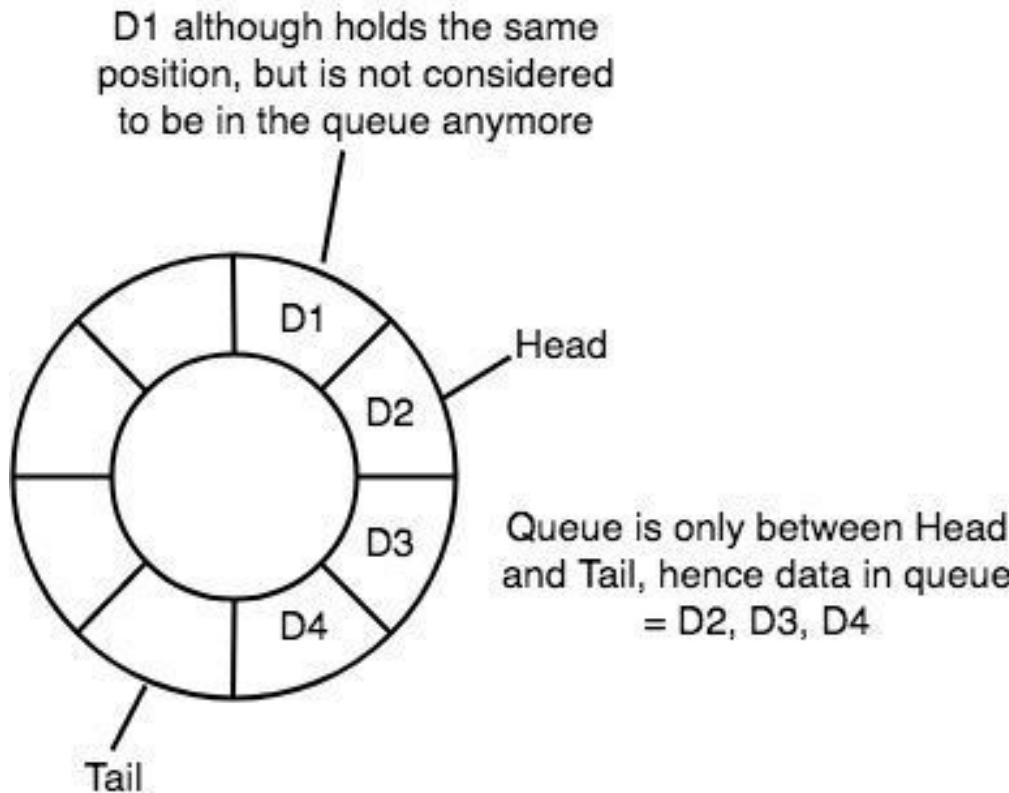
- In case of a circular queue, head pointer will always point to the front of the queue, and tail pointer will always point to the end of the queue.
- Initially, the head and the tail pointers will be pointing to the same location, this would mean that the queue is empty.



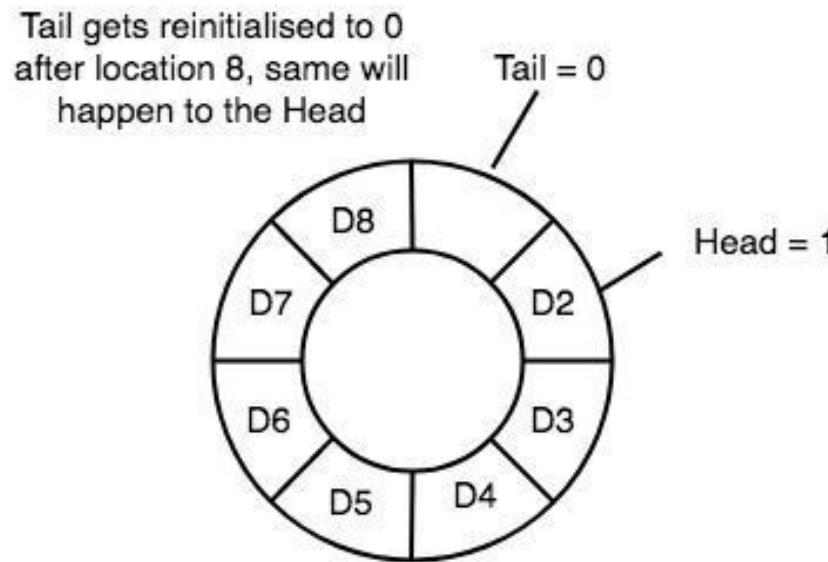
- New data is always added to the location pointed by the tail pointer, and once the data is added, tail pointer is incremented to point to the next available location.



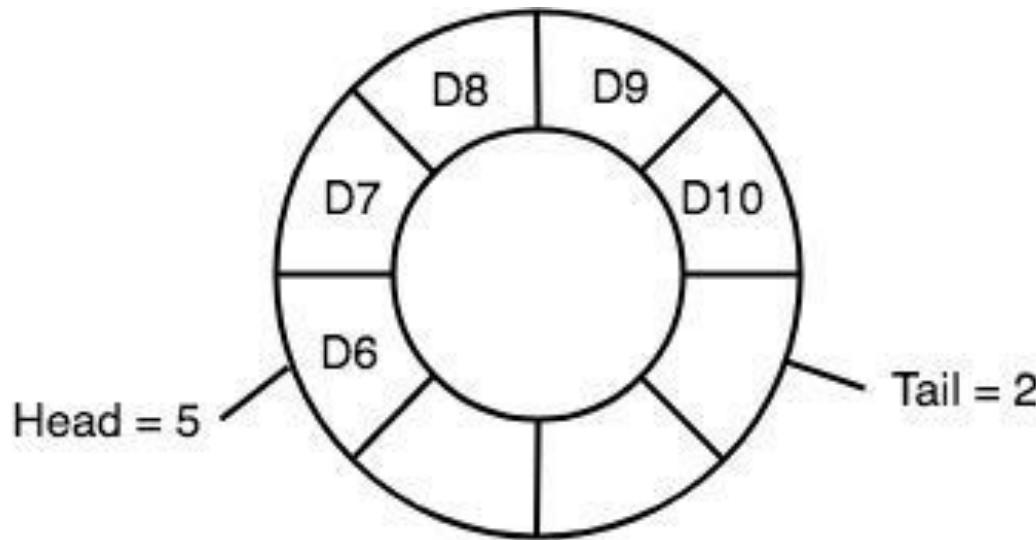
- In a circular queue, data is not actually removed from the queue. Only the head pointer is incremented by one position when **dequeue** is executed. As the queue data is only the data between head and tail, hence the data left outside is not a part of the queue anymore, hence removed.



- The head and the tail pointer will get reinitialised to **0** every time they reach the end of the queue.



- Also, the head and the tail pointers can cross each other. In other words, head pointer can be greater than the tail.
 - How? This will happen when we dequeue the queue a couple of times and the tail pointer gets reinitialised upon reaching the end of the queue.



In such a situation the value of the Head pointer will be greater than the Tail pointer

- the value of the tail and the head pointer must be within the maximum queue size.
- Suppose, If the queue has a size of 8, hence, the value of tail and head pointers will always be between 0 and 7.
- This can be controlled either by checking everytime whether tail or head have reached the `maxSize` and then setting the value 0 or, we have a better way, which is, for a value x if we divide it by 8, the remainder will never be greater than 8, it will always be between 0 and 0, which is exactly what we want.
- So the formula to increment the head and tail pointers to make them go **round and round** over and again will be,
 - $\text{head} = (\text{head}+1) \% \text{maxSize}$
 - $\text{tail} = (\text{tail}+1) \% \text{maxSize}$

Implementation of Circular Queue

Initialize the queue, with size of the queue defined (`maxSize`), and head and tail pointers.

Enqueue():

Check if the number of elements is equal to `maxSize - 1`:

If **Yes**, then return **Queue is full**.

If **No**, then add the new data element to the location of tail pointer and increment the tail pointer.

Dequeue():

Check if the number of elements in the queue is zero:

If **Yes**, then return **Queue is empty**.

If **No**, then increment the head pointer.

Finding the size:

If, **tail >= head**, $\text{size} = (\text{tail} - \text{head}) + 1$

But if, **head > tail**, $\text{size} = \text{maxSize} - (\text{head} - \text{tail}) + 1$

Enqueue

STEP 1: IF FRONT=0 AND REAR = MAX-1

 WRITE “OVERFLOW”

 GOTO STEP 4

STEP 2: IF FRONT=-1 AND REAR = -1

 SET FRONT=REAR=0

 ELSE IF REAR=MAX-1 AND FRONT!=0

 SET REAR=0

 ELSE

 SET REAR = REAR+1

STEP 3: SET QUEUE[REAR] = VAL

STEP 4: EXIT

DEQUEUE

STEP 1: IF FRONT=-1

 WRITE “UNDERFLOW”

 GOTO STEP 4

STEP 2: SET VAL = QUEUE[FRONT]

STEP 3: IF FRONT=REAR //ONLY ONE ELEMENT IN QUEUE

 SET FRONT = REAR = -1

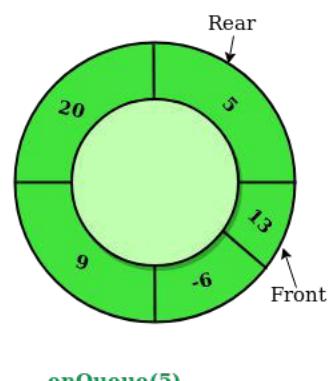
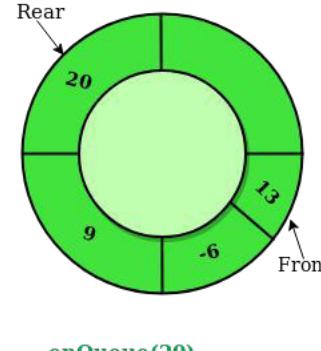
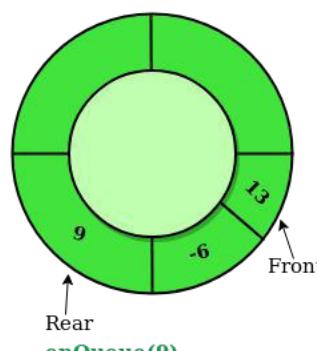
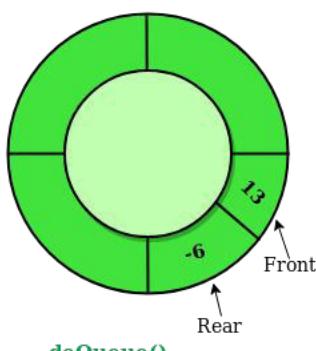
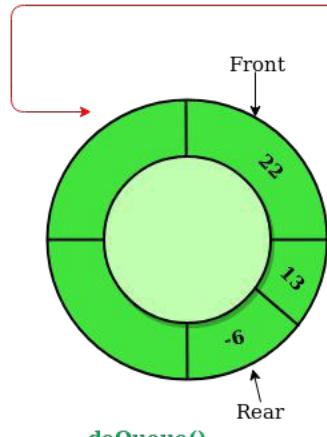
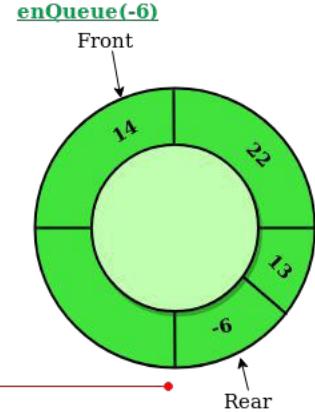
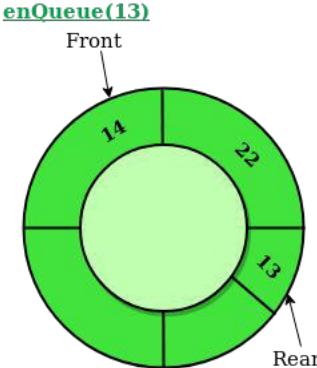
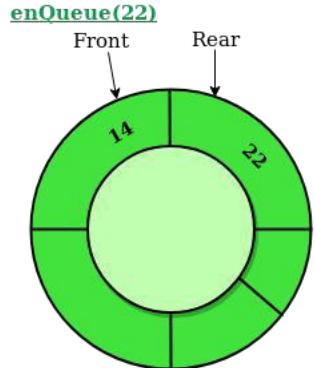
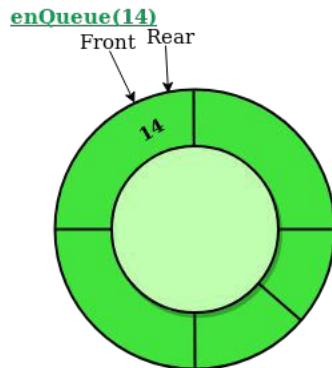
 ELSE IF FRONT = MAX-1 //FRONT HAS REACHED LAST ELEMENT

 SET FRONT = 0

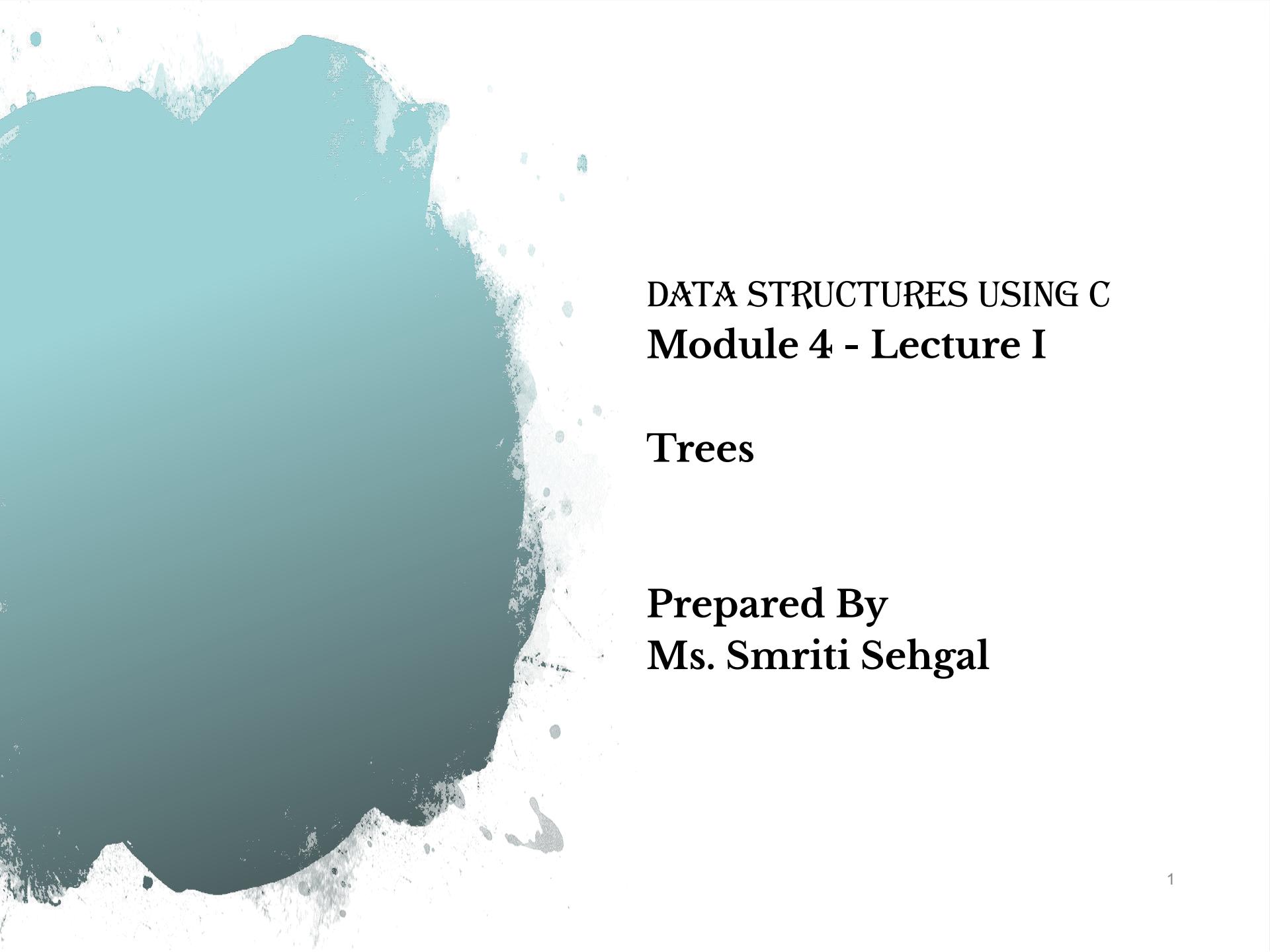
 ELSE

 SET FRONT=FRONT+1

STEP 4: EXIT





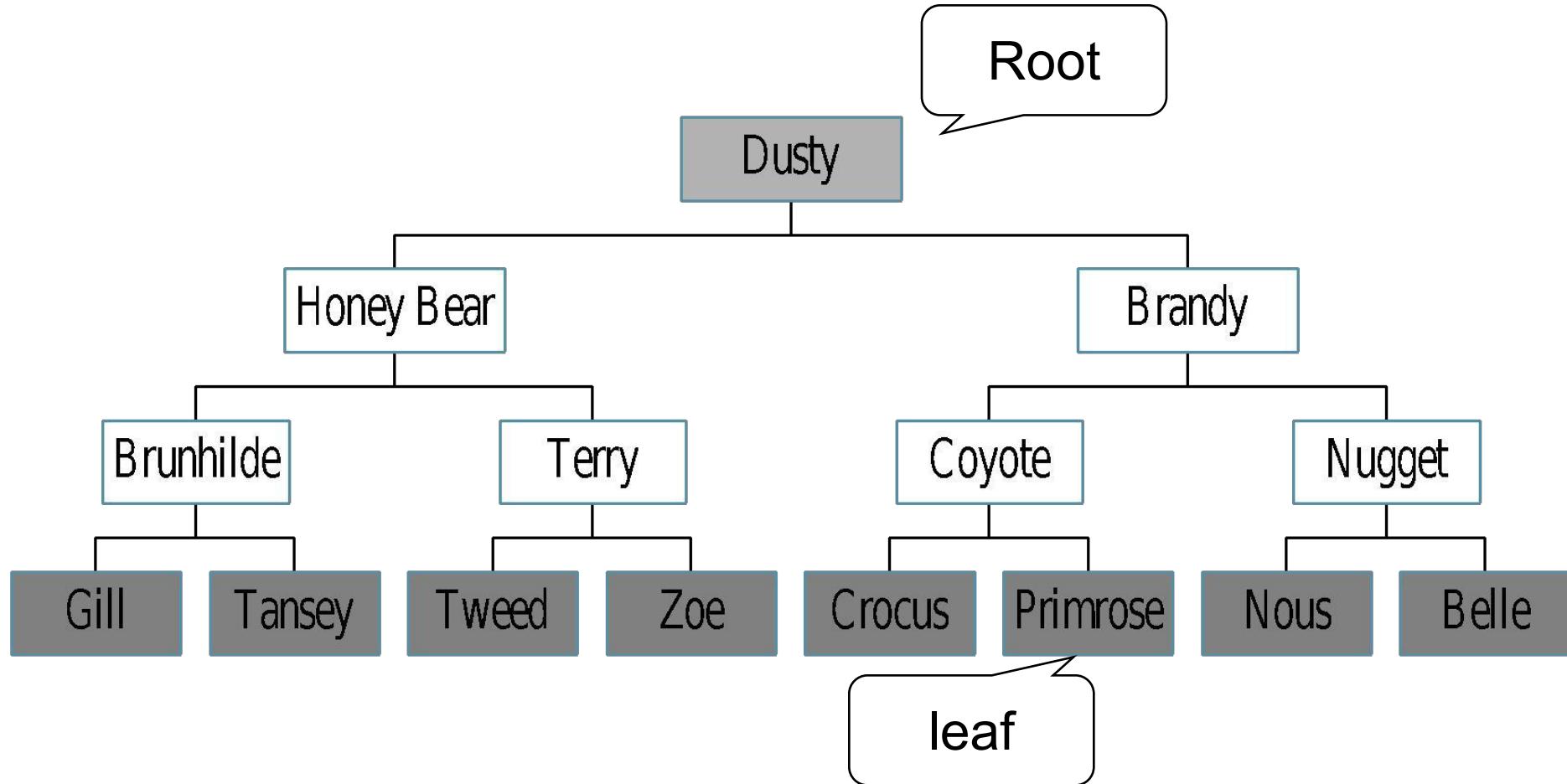


DATA STRUCTURES USING C

Module 4 - Lecture I

Trees

Prepared By
Ms. Smriti Sehgal



- A tree is a finite set of one or more nodes such that:
 - There is a specially designated node called the root.
 - The remaining nodes are partitioned into $n \geq 0$ disjoint sets T_1, \dots, T_n , where each of these sets is a tree.
 - We call T_1, \dots, T_n the subtrees of the root.

- **Path** – Path refers to the sequence of nodes along the edges of a tree.
- **Root** – The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- **Parent** – Any node except the root node has one edge upward to a node called parent.
- **Child** – The node below a given node connected by its edge downward is called its child node.
- **Leaf** – The node which does not have any child node is called the leaf node.
- **Subtree** – Subtree represents the descendants of a node.
- **Visiting** – Visiting refers to checking the value of a node when control is on the node.
- **Traversing** – Traversing means passing through nodes in a specific order.
- **Levels** – Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
- **keys** – Key represents a value of a node based on which a search operation is to be carried out for a node.

Height of a Node

The height of a node is the number of edges from the node to the deepest leaf (ie. the longest path from the node to a leaf node).

Depth of a Node

The depth of a node is the number of edges from the root to the node.

Height of a Tree

The height of a Tree is the height of the root node or the depth of the deepest node.

Degree of a Node

The degree of a node is the total number of branches of that node.

of Nodes = 13

degree of a node (A) = 3

leaf (terminal) = K,L,M,G,I,J

Nonterminal = all others

parent

children

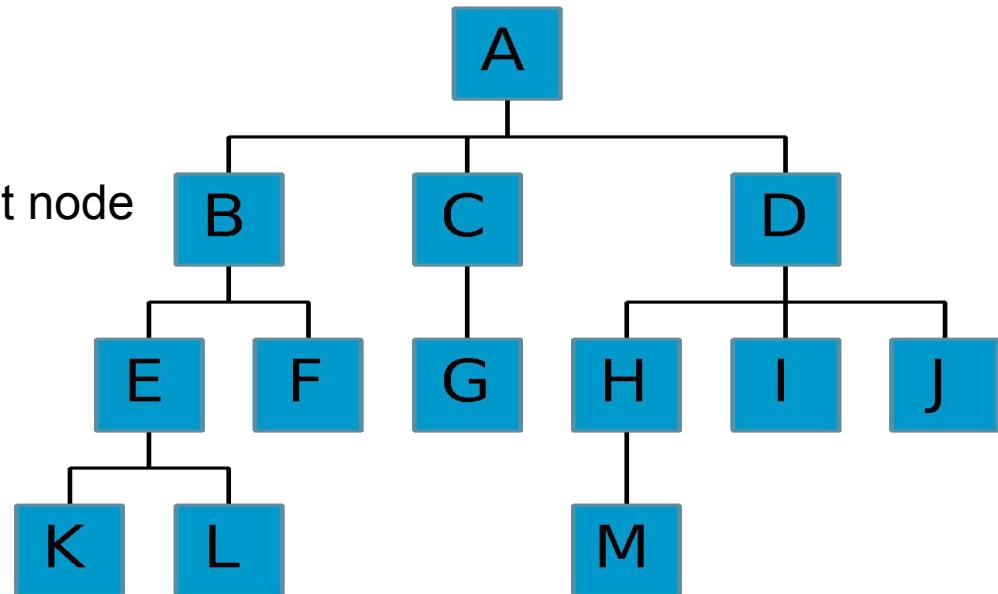
sibling

degree of a tree = degree of root node

ancestor

level of a node

height of a tree = 3

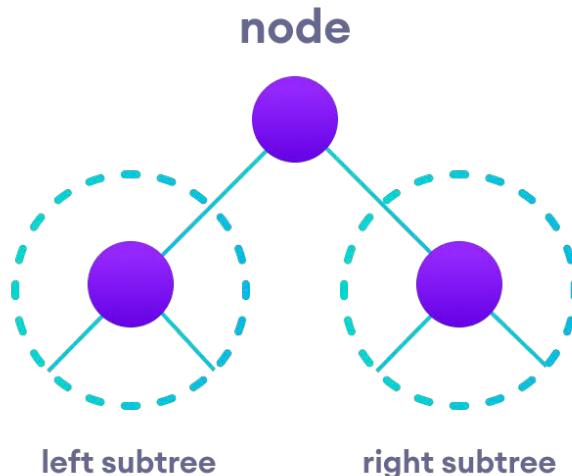


Tree Traversal - inorder, preorder and postorder

Traversing a tree means visiting every node in the tree.

Every tree is a combination
of
• A node carrying data
• Two subtrees

```
struct node
{
    int data;
    struct node* left;
    struct node* right;
}
```



Depending on the order in which we traverse all the nodes, there are three types of traversals.

Inorder traversal

1. First, visit all the nodes in the left subtree
2. Then the root node
3. Visit all the nodes in the right subtree

`inorder(root->left)`
`display(root->data)`
`inorder(root->right)`

Preorder traversal

1. Visit root node
2. Visit all the nodes in the left subtree
3. Visit all the nodes in the right subtree

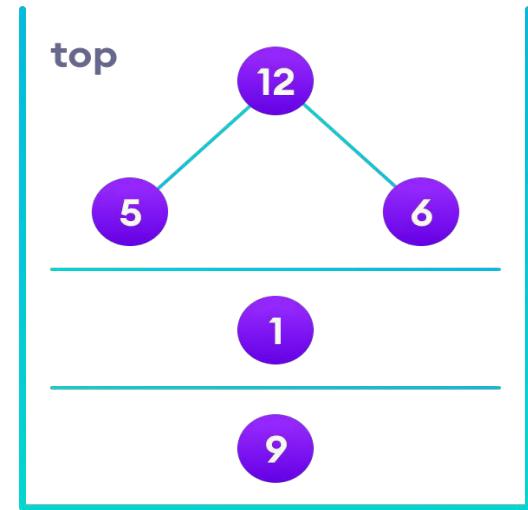
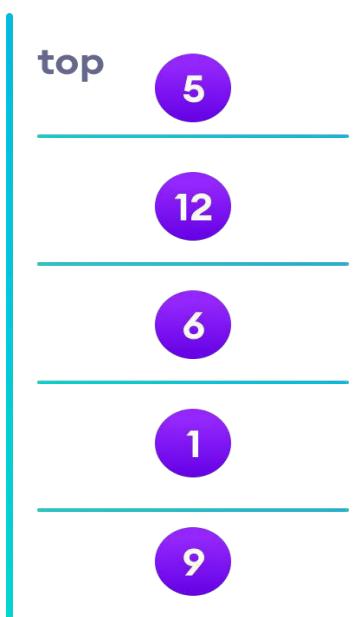
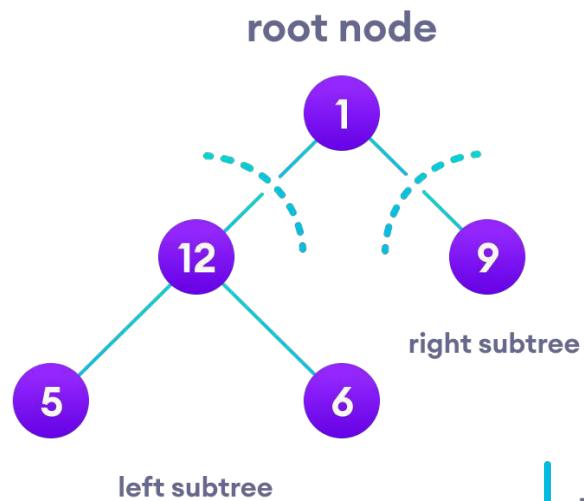
`display(root->data)`
`preorder(root->left)`
`preorder(root->right)`

Postorder traversal

1. Visit all the nodes in the left subtree
2. Visit all the nodes in the right subtree
3. Visit the root node

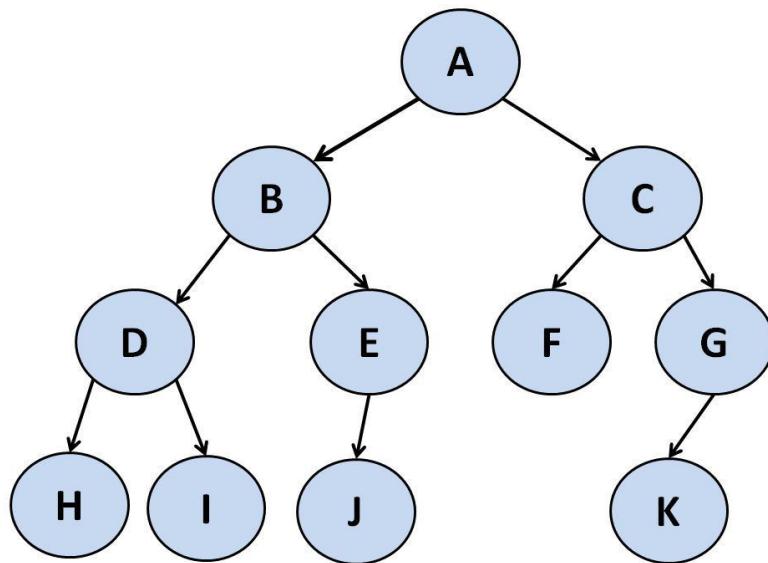
`postorder(root->left)`
`postorder(root->right)`
`display(root->data)`

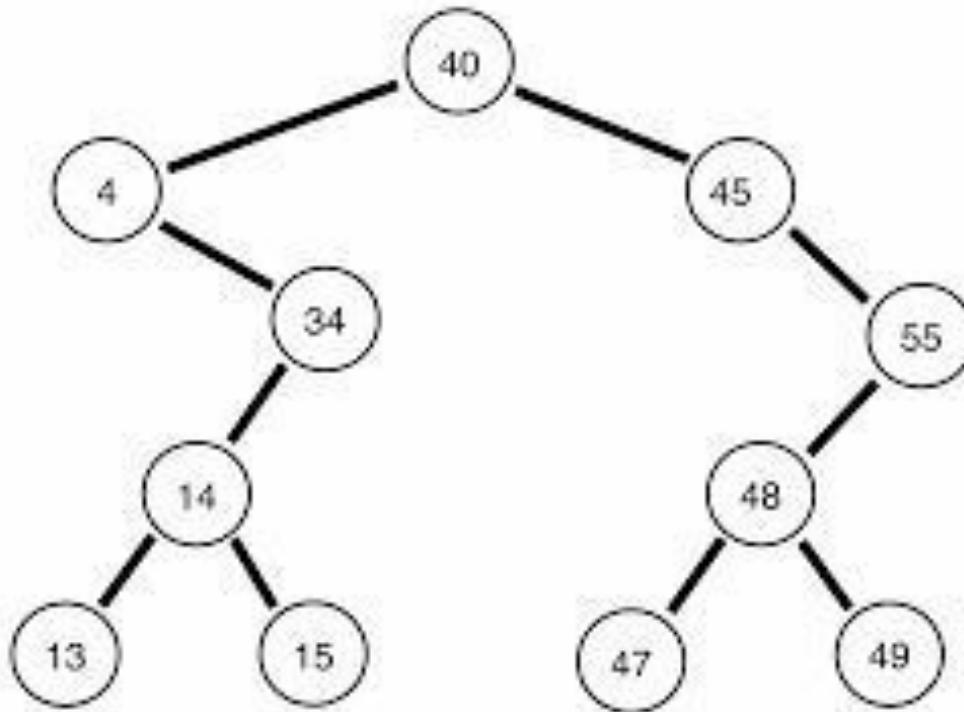
Eg. Inorder



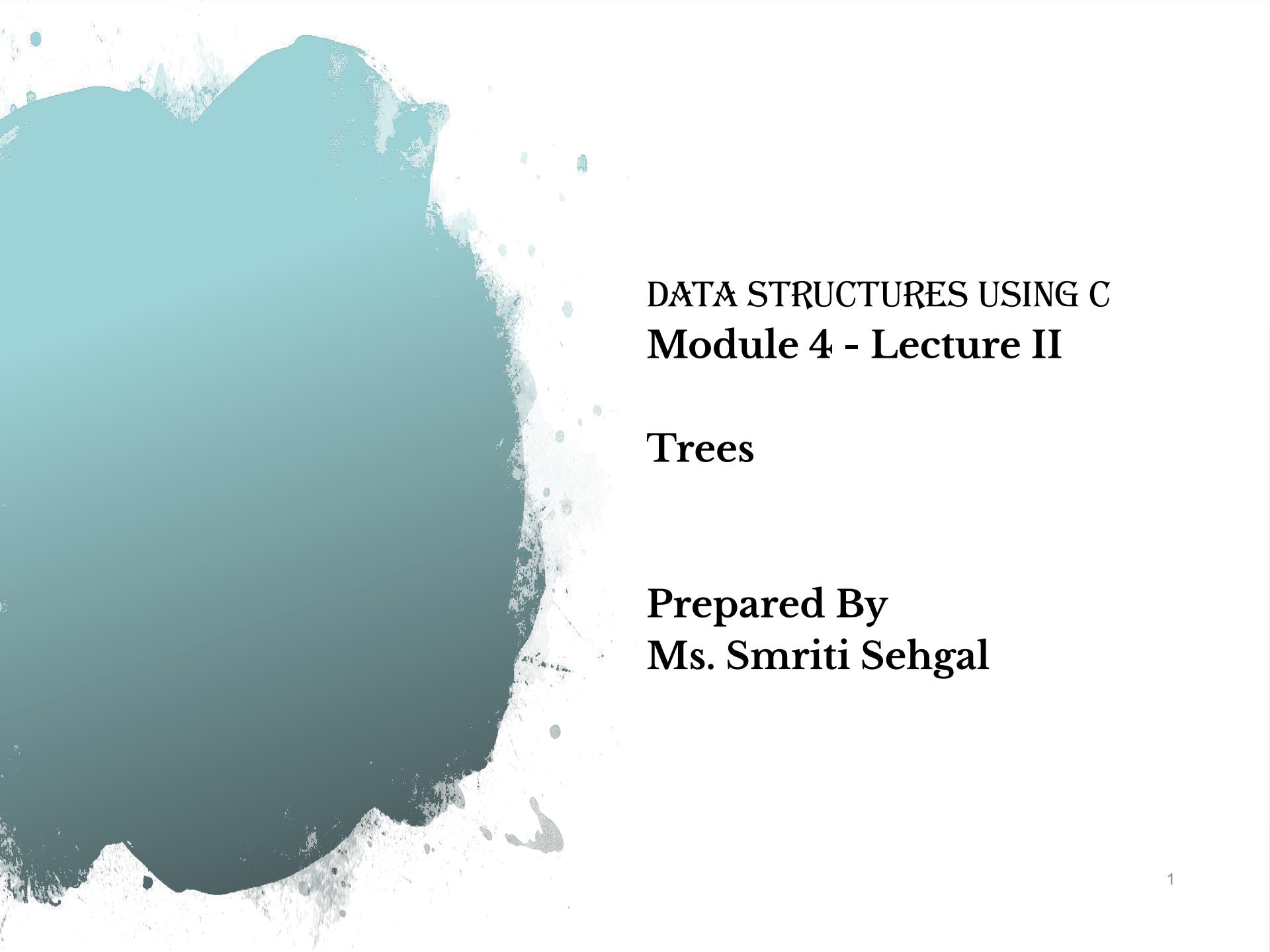
The inorder traversal is as

5 -> 12 -> 6 -> 1 -> 9









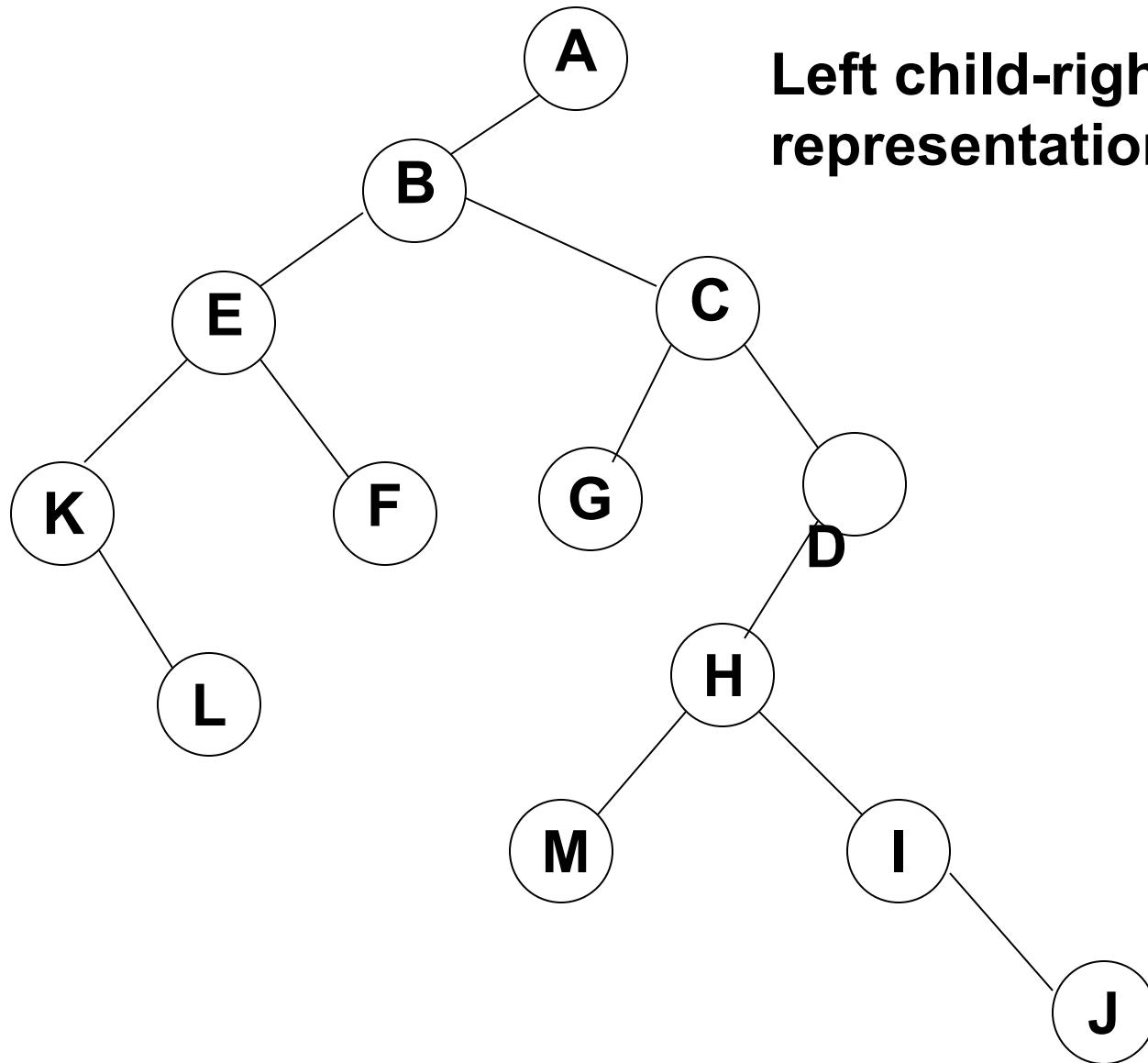
DATA STRUCTURES USING C

Module 4 - Lecture II

Trees

Prepared By
Ms. Smriti Sehgal

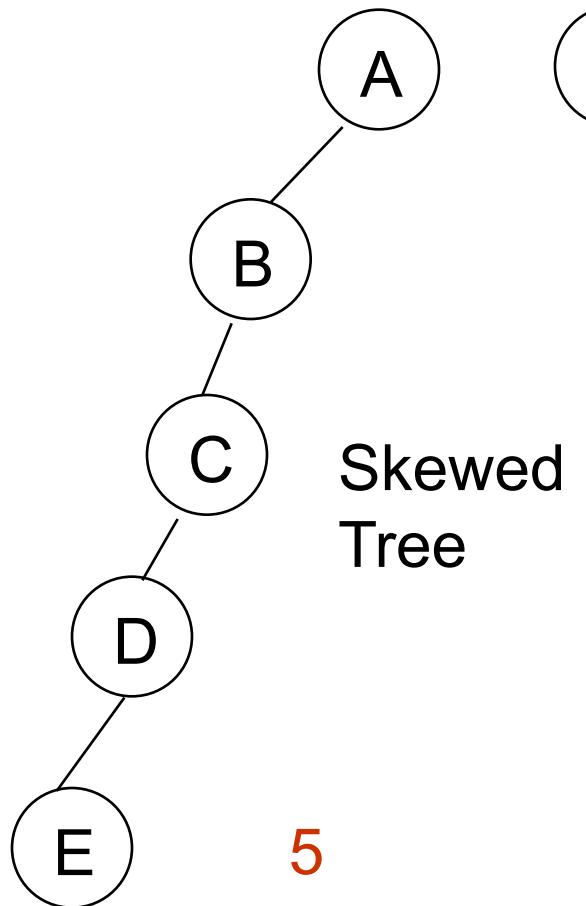
- A binary tree is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called *the left subtree* and *the right subtree*.
- Any tree can be transformed into binary tree.
 - by left child-right sibling representation
- The left subtree and the right subtree are distinguished.



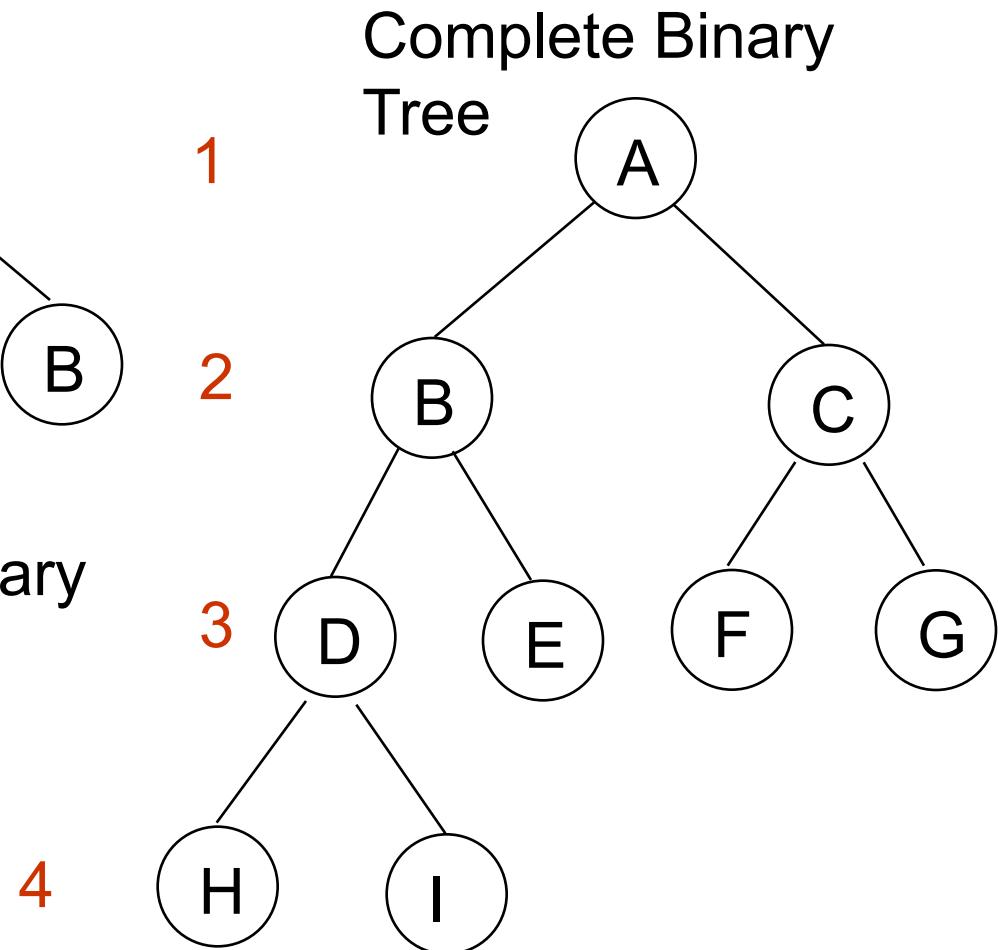
Left child-right child tree representation of a tree

Samples of Trees

Amity School of Engineering & Technology



Skewed Binary
Tree



The maximum number of nodes at level 'l' of a binary tree is 2^l

Level is number of nodes on path from root to the node (including root and node).

Level of root is 0.

This can be proved by induction.

For root, l = 0, number of nodes = $2^0 = 1$

Assume that maximum number of nodes on level 'l' is 2^l

Since in Binary tree every node has at most 2 children, next level would have twice nodes, i.e. $2 * 2^l$

Maximum number of nodes in a binary tree of height 'h' is
$$2^h - 1$$

Height of a tree is maximum number of nodes on, root to leaf path.

Height of a tree with single node is considered as 1.

A tree has maximum nodes if all levels are complete i.e they have maximum nodes.

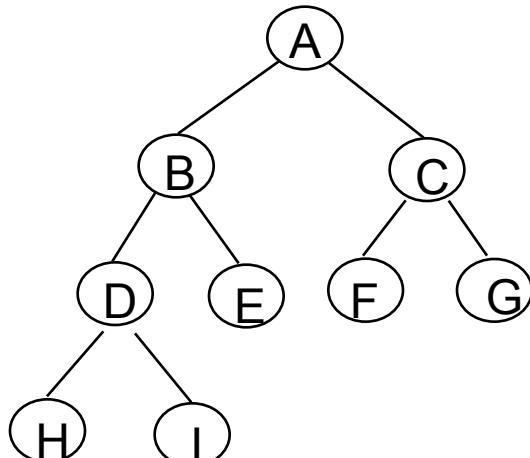
So maximum number of nodes in a binary tree of height h is $1 + 2 + 4 + \dots + 2^{h-1}$.

This is a simple geometric series with h terms and sum of this series is $2^h - 1$.
In some books, height of the root is considered as 0. In this convention, the above formula becomes $2^{h+1} - 1$

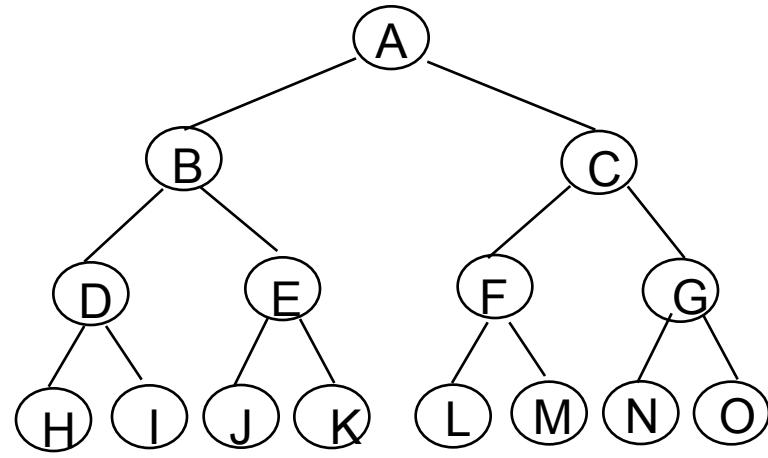
In a Binary Tree with N nodes, minimum possible height or minimum number of levels is ? $\log_2(N+1)$?

In Binary tree where every node has 0 or 2 children, number of leaf nodes is always one more than nodes with two children.

- A **full binary tree** is a **tree** in which every node other than the leaves has two children..
- A **complete binary tree** is a BT in which all levels except last one is completely filled and all nodes appear as far left as possible.



Complete binary tree



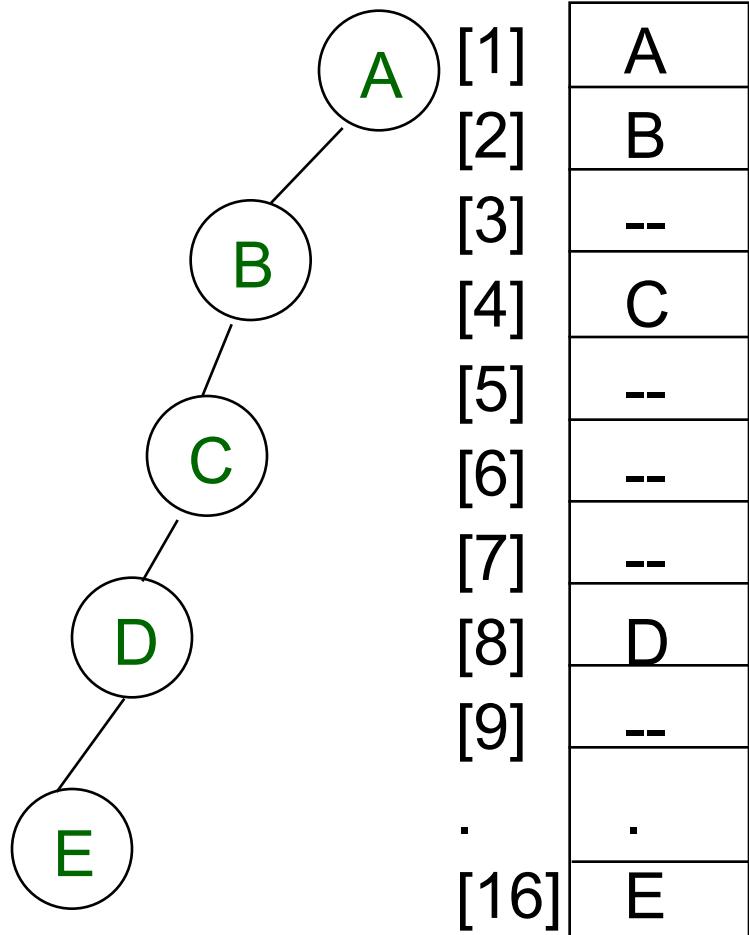
Full binary tree of depth 4

Binary Tree Representations

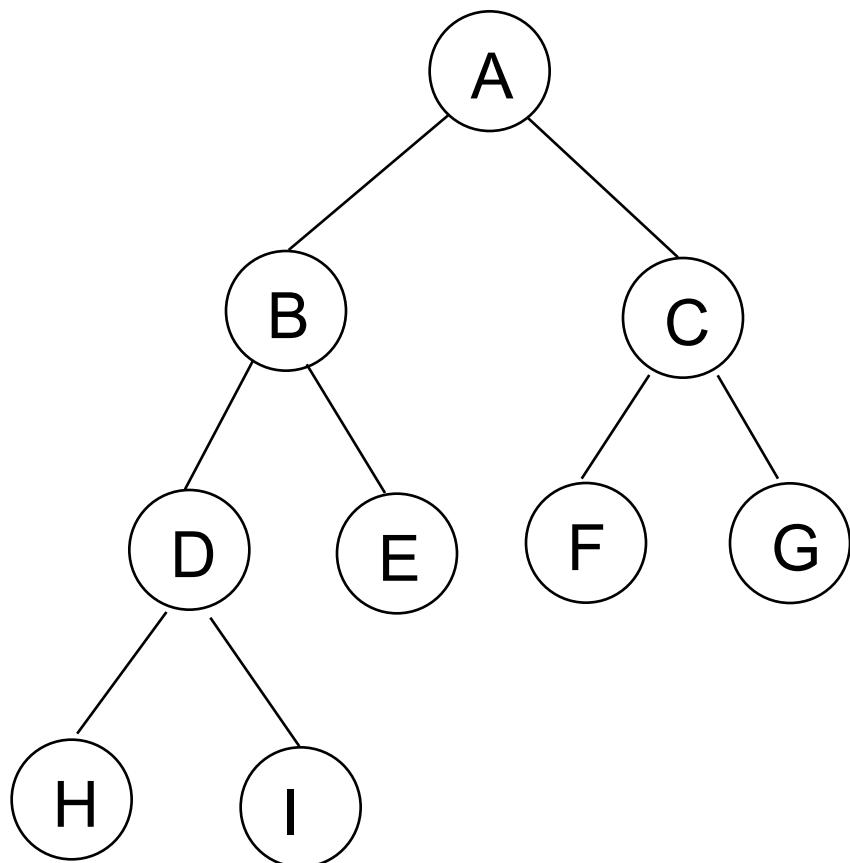
- If a complete binary tree with n nodes is represented sequentially, then for any node with index i , $1 \leq i \leq n$, we have:
 - $\text{parent}(i)$ is at $i/2$ if $i \neq 1$. If $i=1$, i is at the root and has no parent.
 - $\text{left_child}(i)$ is at $2i$ if $2i \leq n$. If $2i > n$, then i has no left child.
 - $\text{right_child}(i)$ is at $2i+1$ if $2i+1 \leq n$. If $2i+1 > n$, then i has no right child.

Sequential Representation

Amity School of Engineering & Technology



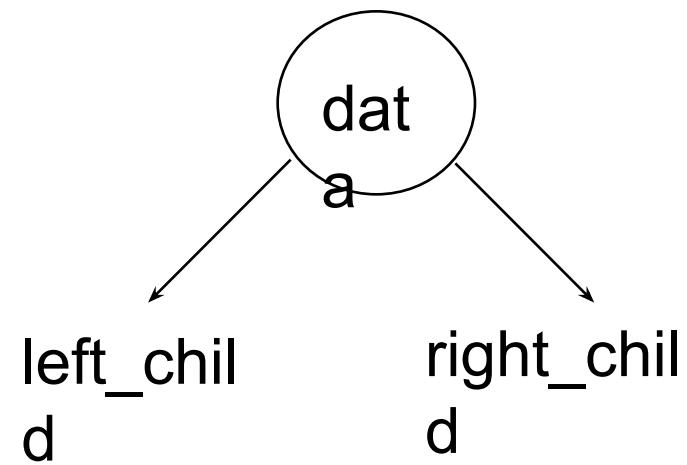
- (1) waste space**
- (2)**
- insertion/deletion problem**



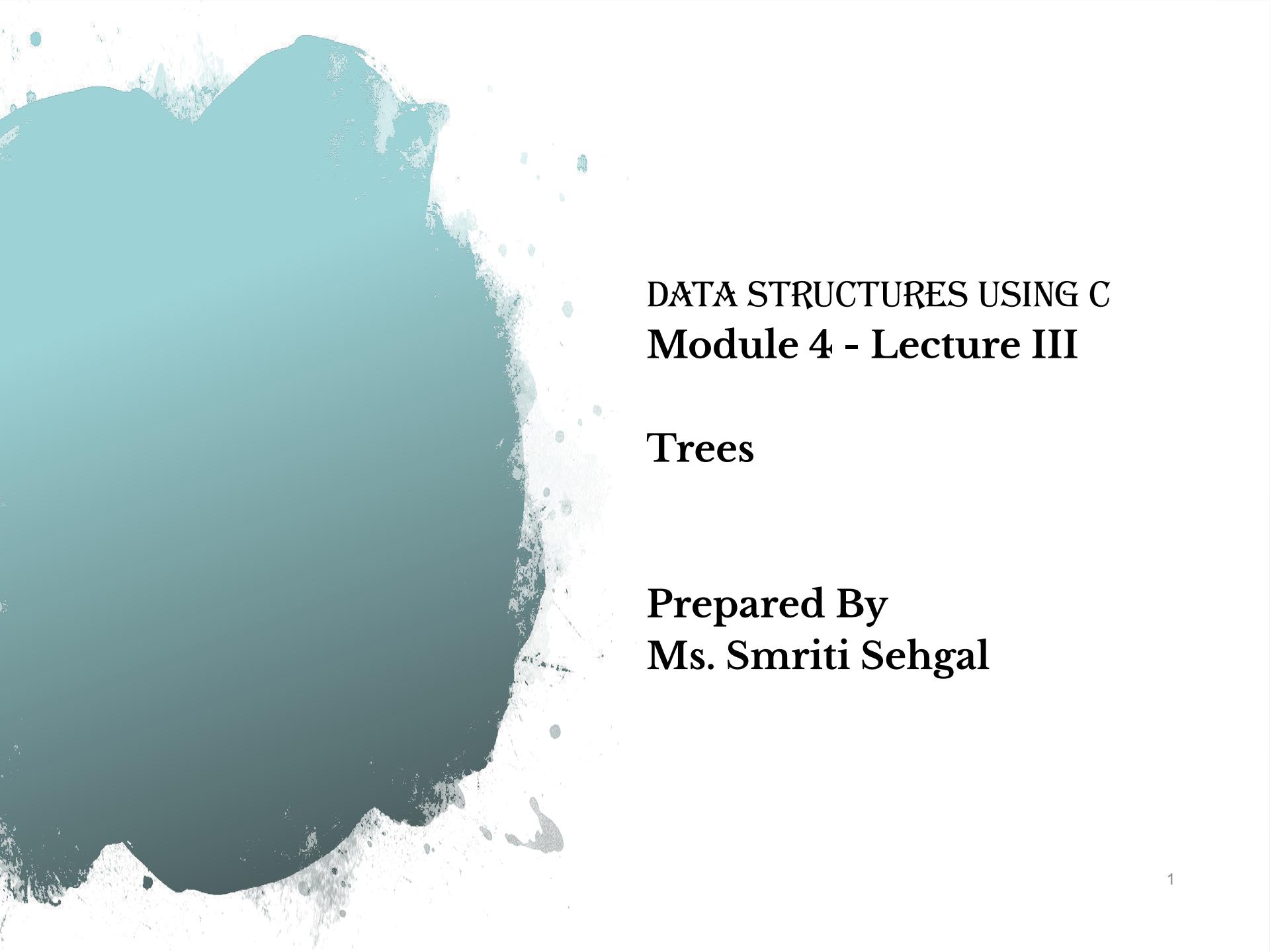
[1]	A
[2]	B
[3]	C
[4]	D
[5]	E
[6]	F
[7]	G
[8]	H
[9]	I

```
struct node {  
    int data;  
    struct node *left_child, *right_child;  
};
```

left_chil	dat	right_chil
d	a	d







DATA STRUCTURES USING C

Module 4 - Lecture III

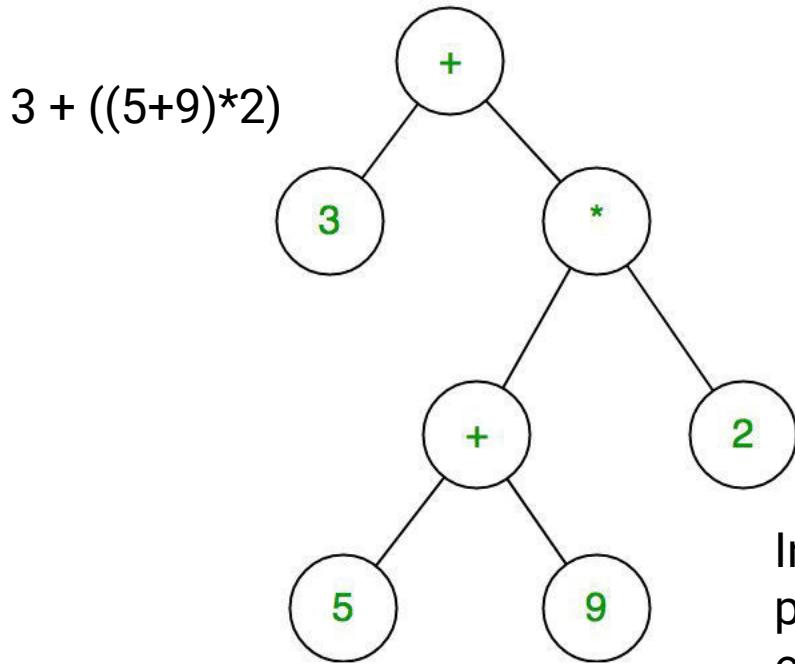
Trees

Prepared By
Ms. Smriti Sehgal

Expression Tree

Amity School of Engineering & Technology

Expression tree is a binary tree in which each internal node corresponds to operator and each leaf node corresponds to operand



Inorder traversal of expression tree produces infix version of given postfix expression (same with preorder traversal it gives prefix expression)

Evaluating the expression represented by expression tree

Let t be the expression tree

If t is not null then

 If t.value is operand then

 Return t.value

 A = solve(t.left)

 B = solve(t.right)

 Return calculate(A, B, t.value)

Construction of Expression Tree

Loop through input expression and do following for every character.

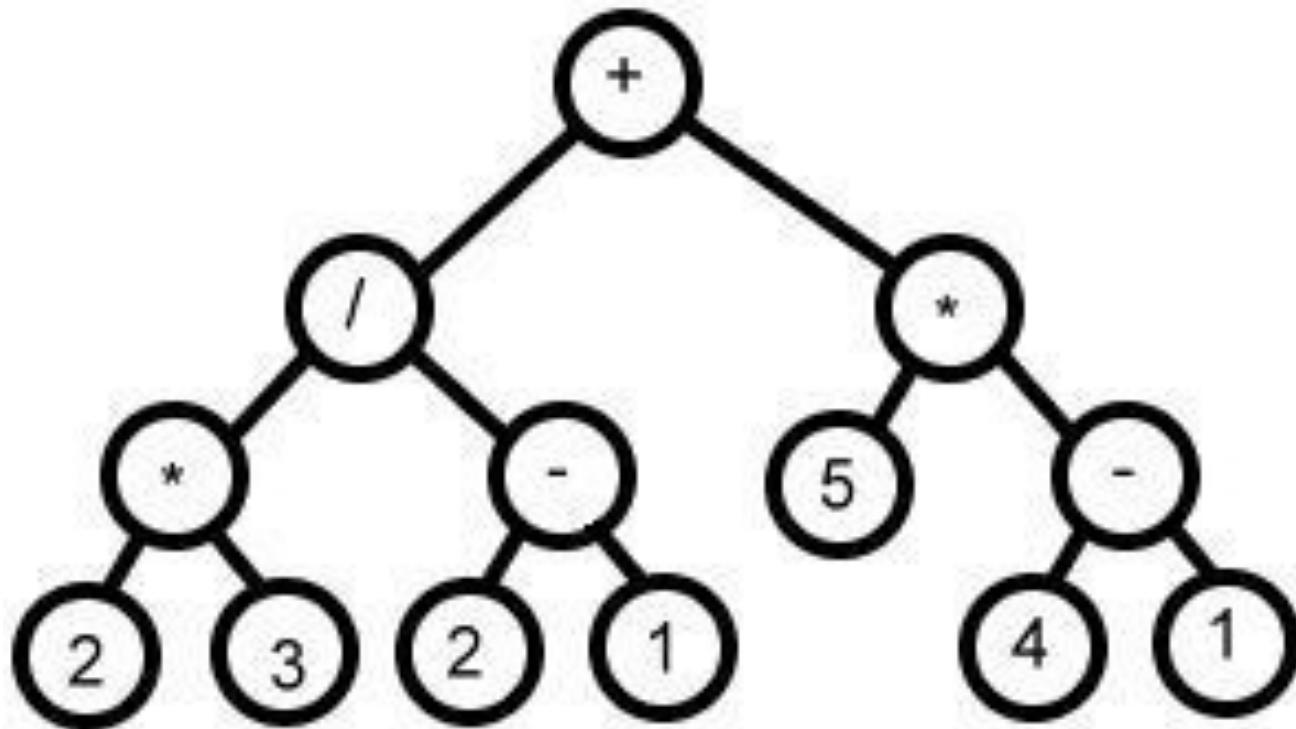
1) If character is operand push that into stack

2) If character is operator pop two values from stack make them its child and push current node again.

At the end only element of stack will be root of expression tree.

Expression Tree Examples

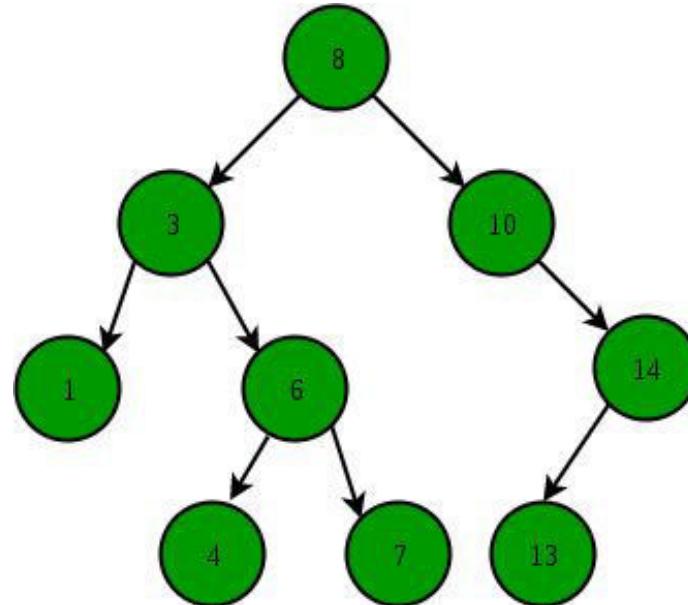
Expression	Expression Tree	Inorder Traversal Result
$(a+3)$	<pre> graph TD plus((+)) --> a((a)) plus --> three((3)) </pre>	$a + 3$
$3+(4*5-(9+6))$	<pre> graph TD plus1((+)) --> three((3)) plus1 --> minus((-)) minus --> mult((*)) minus --> plus2((+)) mult --> four((4)) mult --> five((5)) plus2 --> nine((9)) plus2 --> six((6)) </pre>	$3+4*5-9+6$
$\log(x)$	<pre> graph TD log((log)) --> x((x)) </pre>	$\log x$
$n!$	<pre> graph TD exclamation((!)) --> n((n)) </pre>	$n !$



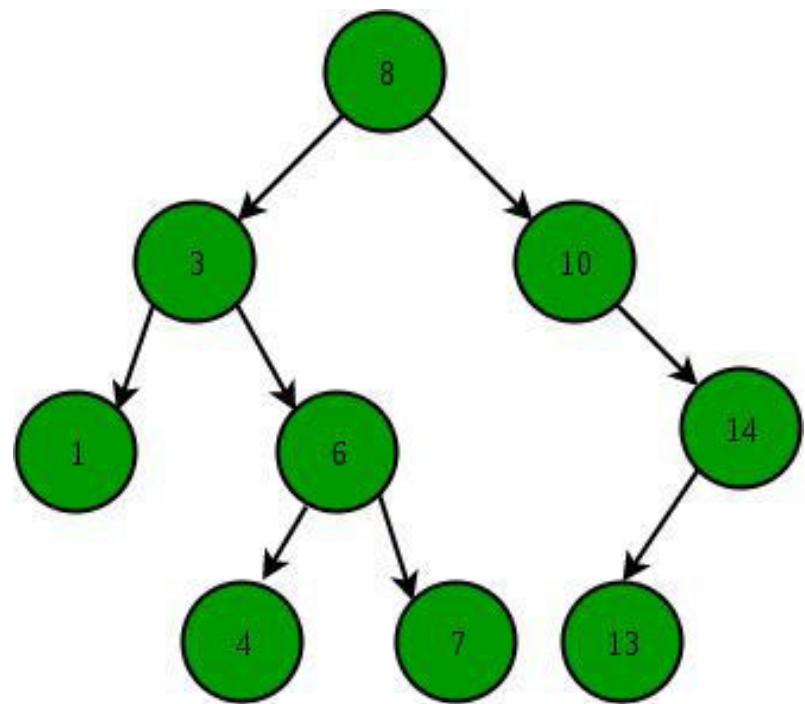
Expression tree for $2*3/(2-1)+5*(4-1)$

Binary Search Tree, is a node-based binary tree data structure which has the following properties:

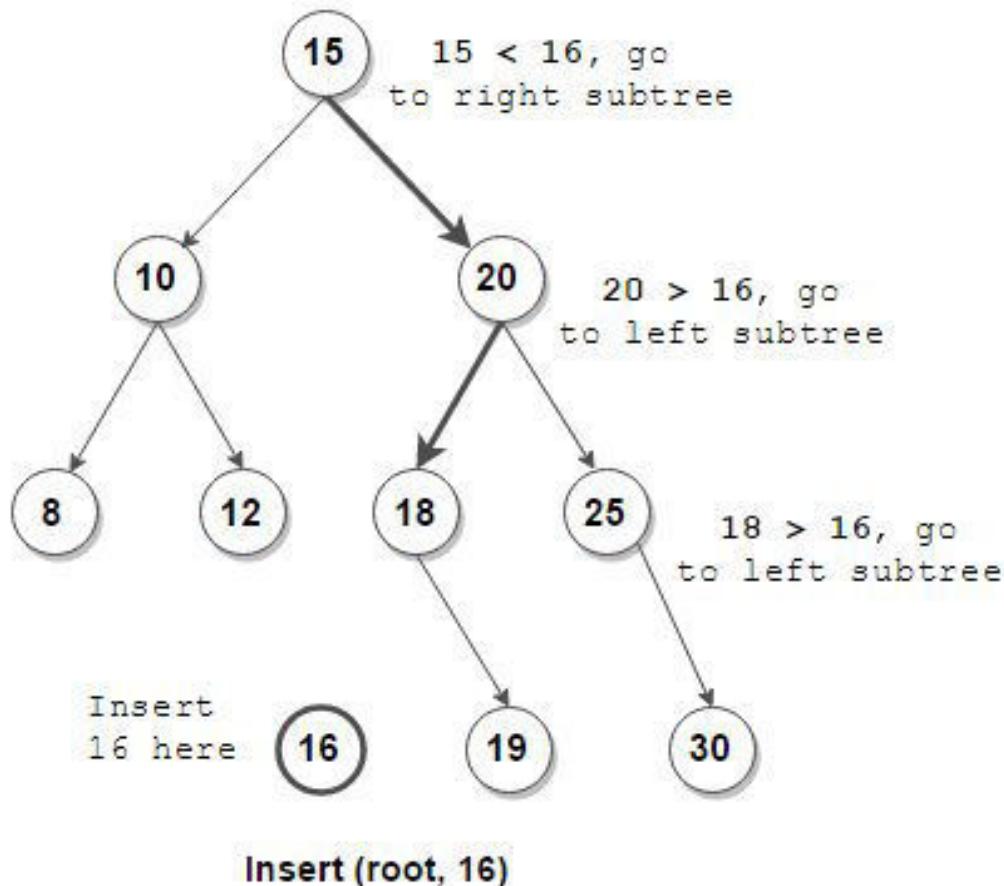
- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.
- There must be no duplicate nodes.



1. Start from root.
2. Compare the inserting element with root,
if less than root, then recurse for left, else
recurse for right.
3. If element to search is found anywhere,
return true, else return false.

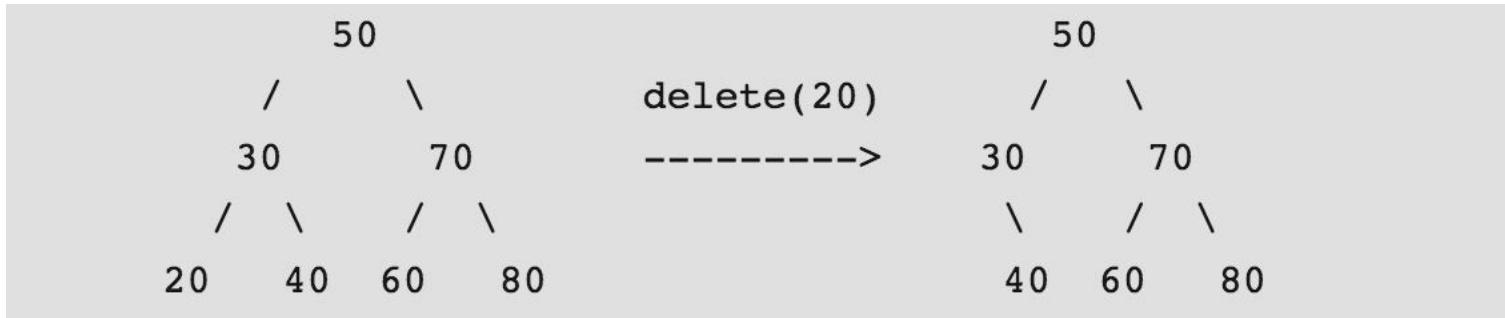


A new key is always inserted at leaf. Search a key from root till we hit a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node.



The worst case time complexity of search and insert operations is $O(h)$ where h is height of Binary Search Tree.

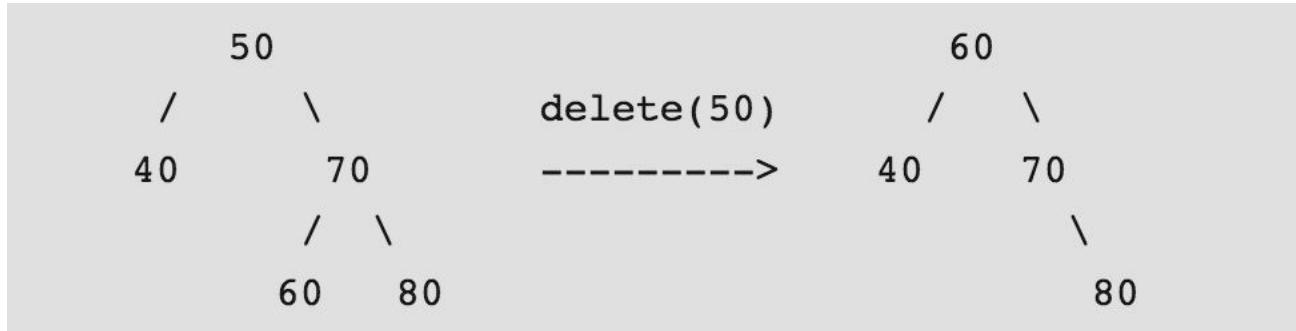
1) Node to be deleted is leaf: Simply remove from the tree.



2. Node to be deleted has only one child: Copy the child to the node and delete the child



3) Node to be deleted has two children: Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.



Create a BST from following:

98,2,48,12,56,32,4,67,23,87,123,55,46

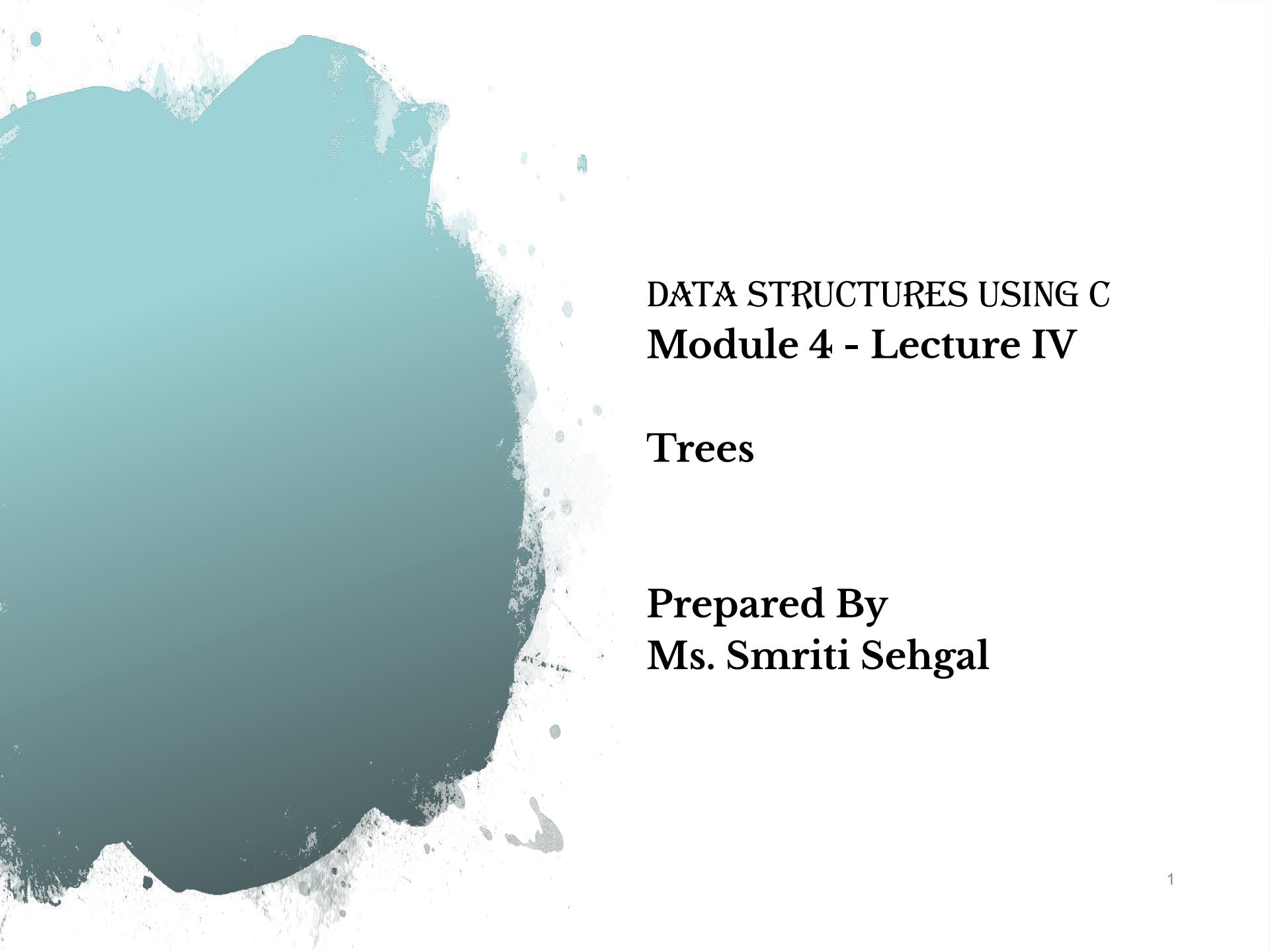
1. Insert 21,39,45,54,63
2. Delete values 23, 56, 2, 45

Create a BST from following:

4,3,8,9,1,2,7,6,10,15,12,99,120,400

1. Insert 312,23,0,99,500
2. Delete values 10,0,7





DATA STRUCTURES USING C

Module 4 - Lecture IV

Trees

Prepared By
Ms. Smriti Sehgal

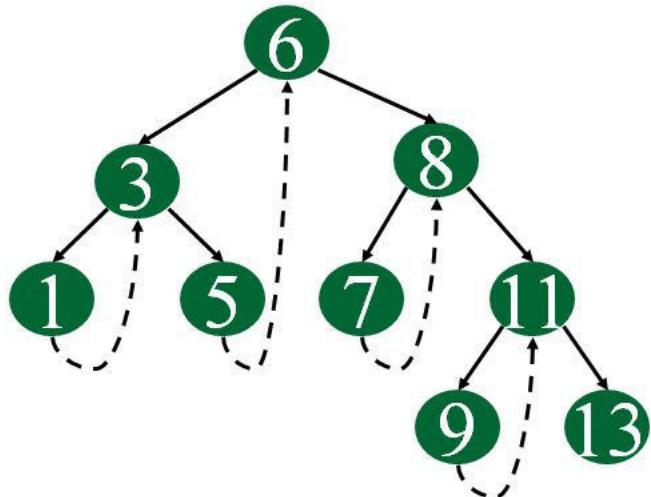
Threaded Tree

- In a linked representation of binary tree, null links can be replaced by pointers, called ‘threads’ to other nodes.
- A left null link of the node is replaced with the address of its inorder predecessor.
- A right null link of node is replaced with the address of its inorder successor.

Threaded Tree

- Binary trees have a lot of wasted space: the leaf nodes each have two null pointers.
- We can use these pointers to help us in inorder traversals.
- We have the pointers reference the next node in an inorder traversal; called ‘threads’.
- We need to know if a pointer is an actual link or a thread, so we keep a Boolean for each pointer.

Single Threaded: Where a NULL right pointers is made to point to the inorder successor (if successor exists)



```

struct Node
{
    int data;
    struct Node *left, *right;
    bool rightThread;
}
  
```

Double Threaded: Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.

```
void inOrder(struct Node *root)
{
    struct Node *cur = leftmost(root);
    while (cur != NULL)
    {
        printf("%d ", cur->data);
        if (cur->rightThread)
            cur = cur->right;
        else
            cur =
leftmost(cur->right);
    }
}
```

```
struct Node* leftMost(struct Node *n)
{
    if (n == NULL)
        return NULL;

    while (n->left != NULL)
        n = n->left;

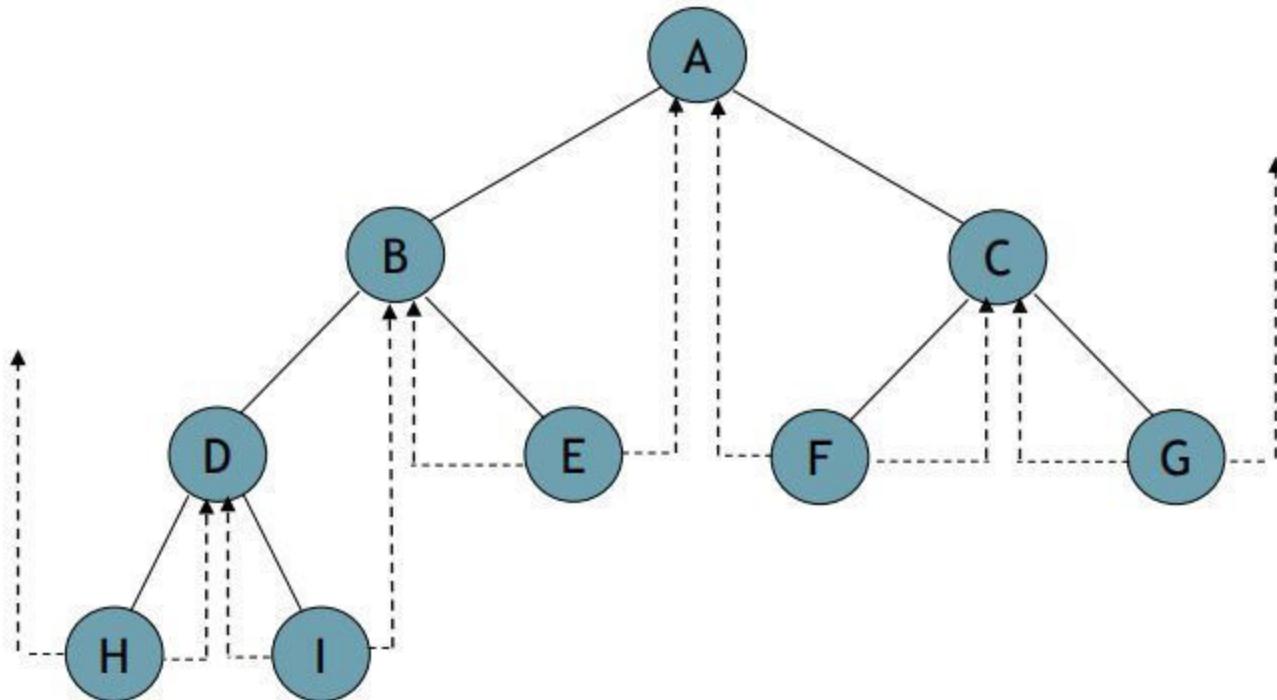
    return n;
}
```

Double Threaded Binary Tree

Threading Rules:

- Right Child field at node ‘x’ is replaced by a pointer to the node that would be visited after ‘x’ when traversing the tree in inorder. That means, it is replaced by the inorder successor of ‘x’.
- Left Child link at node ‘x’ is replaced by a pointer to the node that immediately precedes node ‘x’ in inorder (i.e., it is replaced by the inorder predecessor of ‘x’).

Threaded Tree Diagram



Inorder Sequence: H, D, I, B, E, A, F, C, G

Threads

To distinguish between normal pointers and threads, two boolean fields, ‘LeftThread’ and ‘RightThread’ are added to the record in memory representation.

- $t \rightarrow \text{LeftChild} = \text{true}$

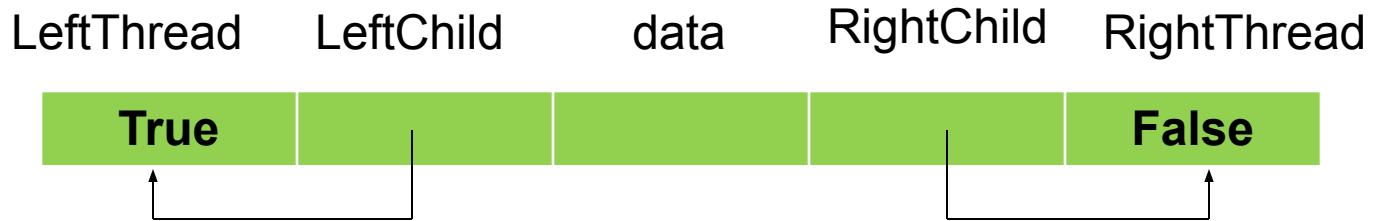
$\Rightarrow t \rightarrow \text{LeftChild}$ is a thread

- $t \rightarrow \text{LeftChild} = \text{false}$

$\Rightarrow t \rightarrow \text{LeftChild}$ is a pointer to the left child.

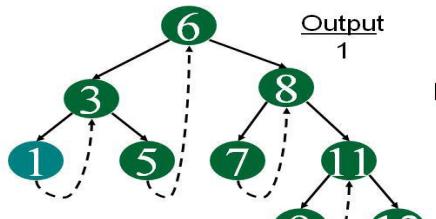
Threads

- To avoid dangling threads, a head node is used in representing a binary tree.
- The original tree becomes the left subtree of the head node.

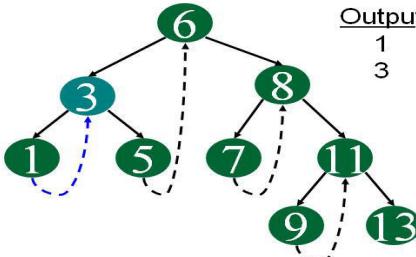


Traversal (Threaded Tree)

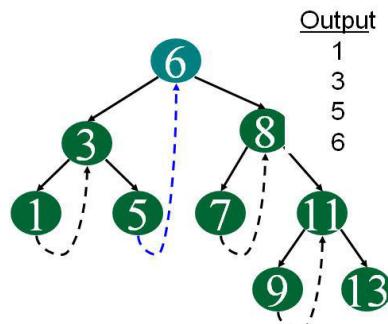
- Start at the leftmost node in the tree, print it, and follow its right thread.
- If we follow a thread to the right, we output the node and continue to its right.
- If we follow a link to the right, we go to the leftmost node, print it, and continue.



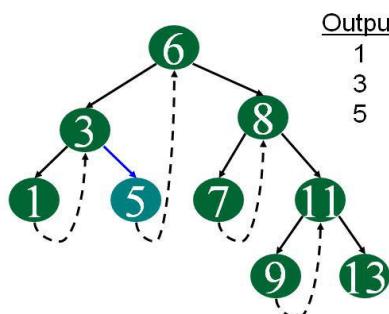
Start at leftmost node, print it



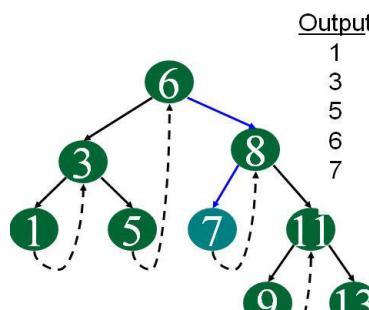
Follow thread to right, print node



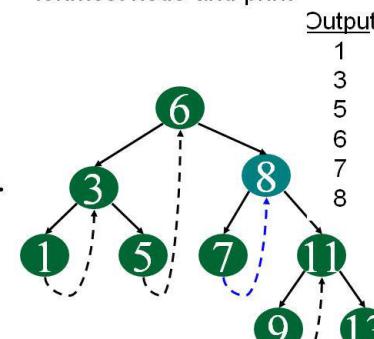
Follow thread to right, print node



Follow link to right, go to leftmost node and print



Follow link to right, go to leftmost node and print

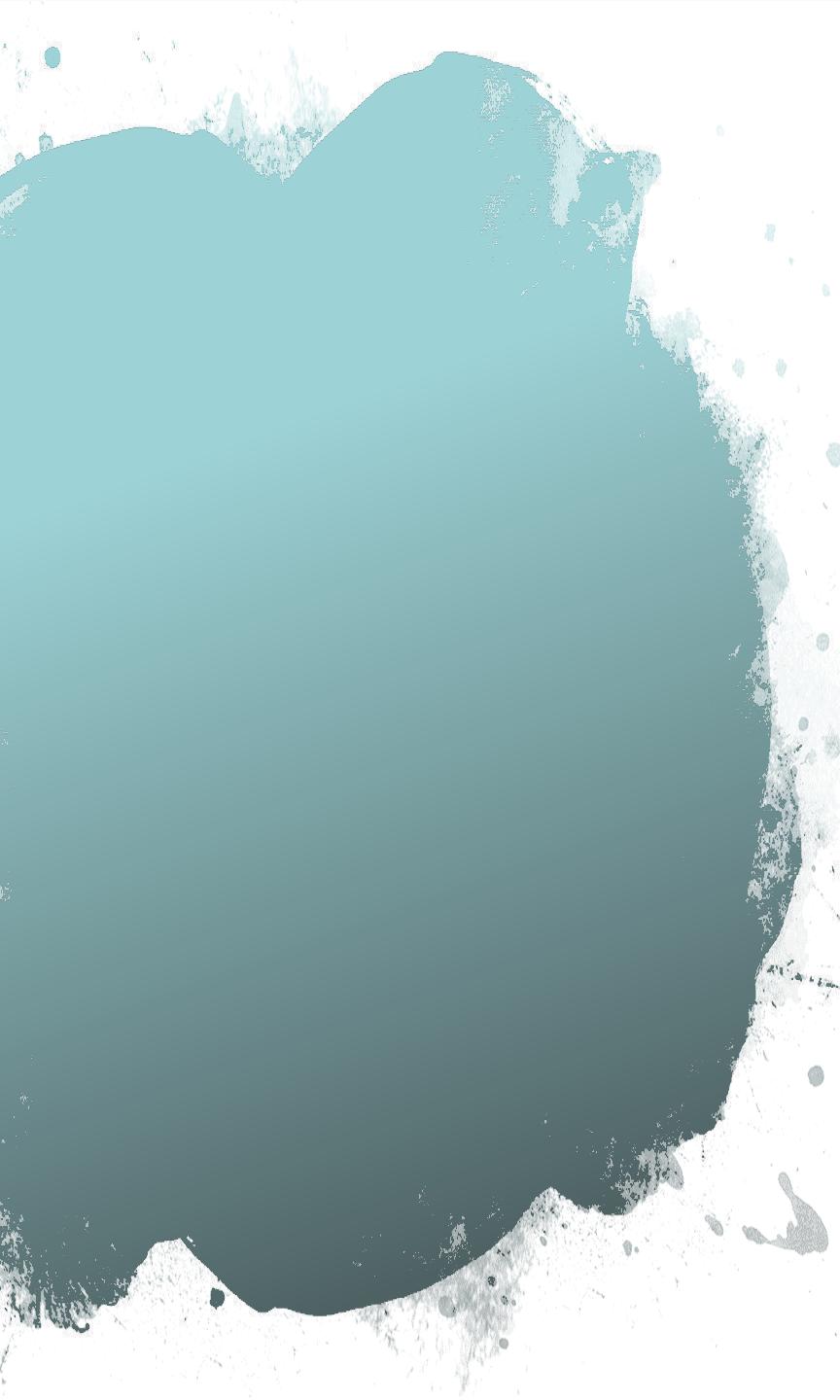


Follow thread to right, print node

Advantages (Threaded Binary Tree)

- Non-recursive preorder traversal can be implemented without a stack.
- Non-recursive inorder traversal can be implemented without a stack.
- Non-recursive postorder traversal can be implemented without a stack.





DATA STRUCTURES USING C

Module 4 - Lecture V

Trees

Prepared By
Ms. Smriti Sehgal

AVL Tree

- It is a **binary search tree** in which the difference of heights of left and right subtrees of any node is less than or equal to **one**.
- Adelson (A), Velskii (V), and Landi (L) developed technique of balancing the **height** of binary trees.

AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either -1, 0 or +1.

Balance factor = (height of left subtree) – (height of right subtree)

It can be defined as:

Let ‘T’ be a **non-empty binary tree** with T_L (left subtrees) and T_R (right subtrees).

The tree is **height balanced** if:

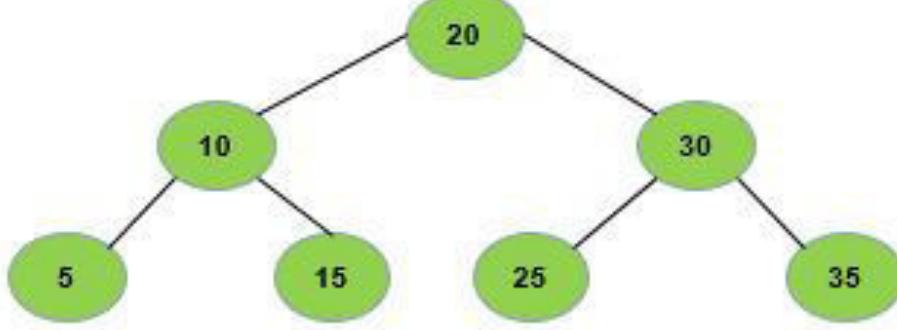
- T_L and T_R are height balanced
- $H_L - H_R \leq 1$, where H_L and H_R are the heights of T_L and T_R
- The Balance factor of a node in a binary tree can have value **1, -1, 0**.
- These values depend on whether the height of its T_L is greater, less than or equal to the height of T_R .

What are the advantages of AVL tree?

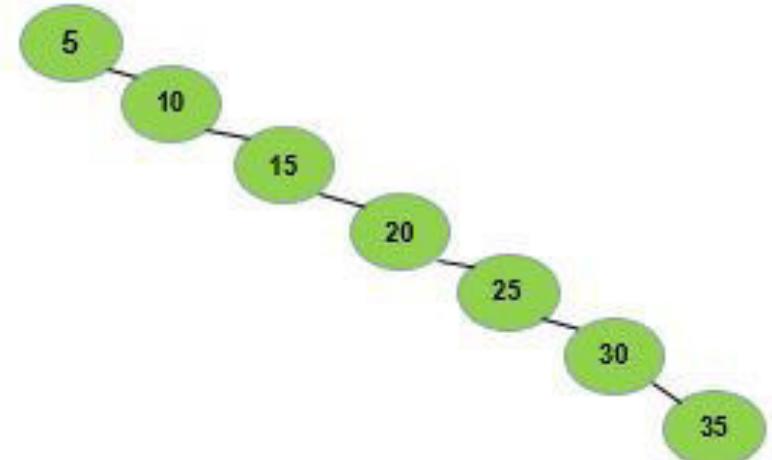
- Low time complexity

Example:

Let a tree having keys 5, 10, 15, 20, 25, 30, 35, then the trees will be like:



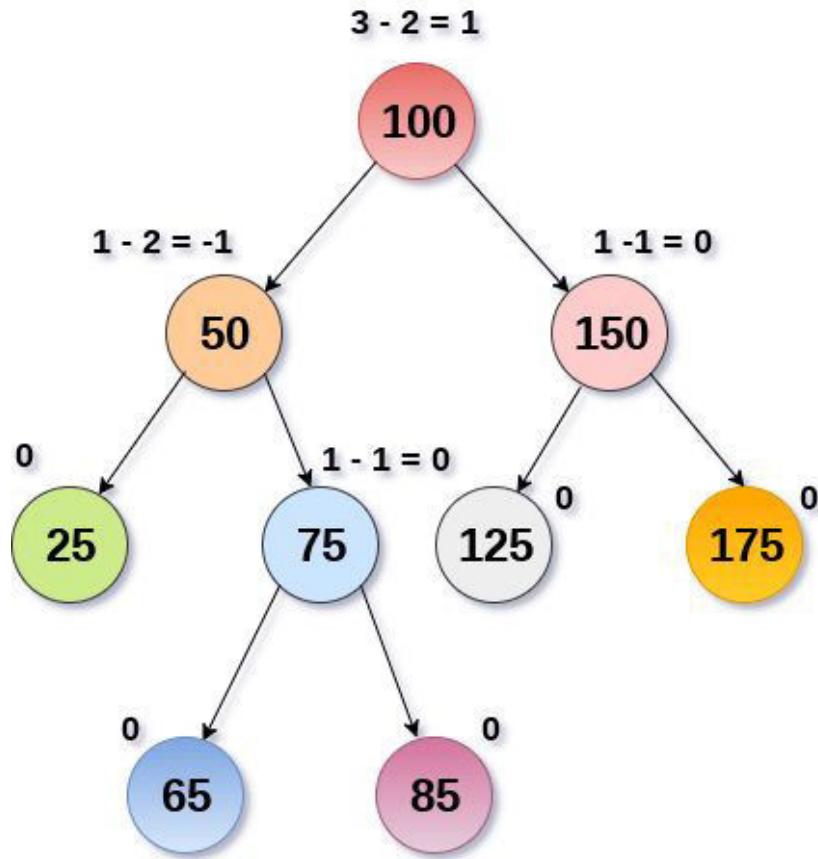
Tree 1



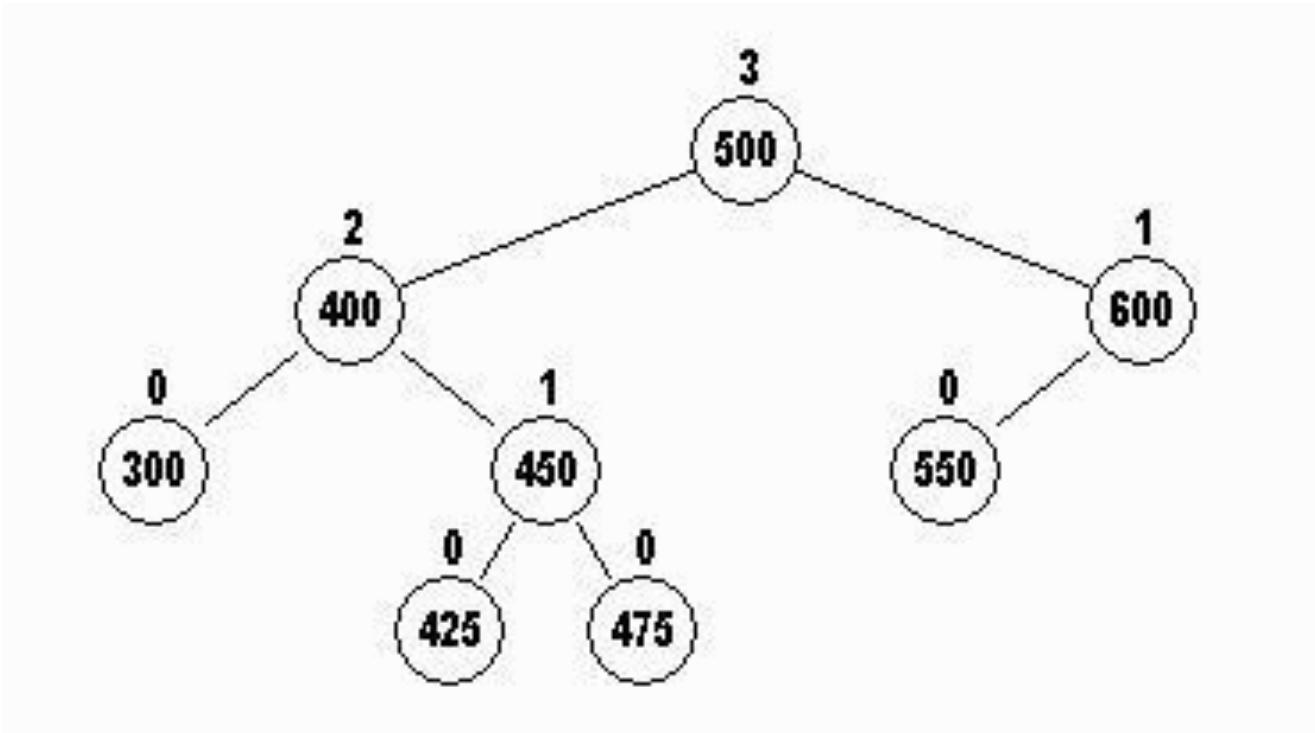
Tree 2

Now, to **insert** a node in the **binary tree**, the algorithm requires seven comparisons (in case of tree-2),

But insertion of the same key in **AVL tree**, algorithm will require three comparisons.

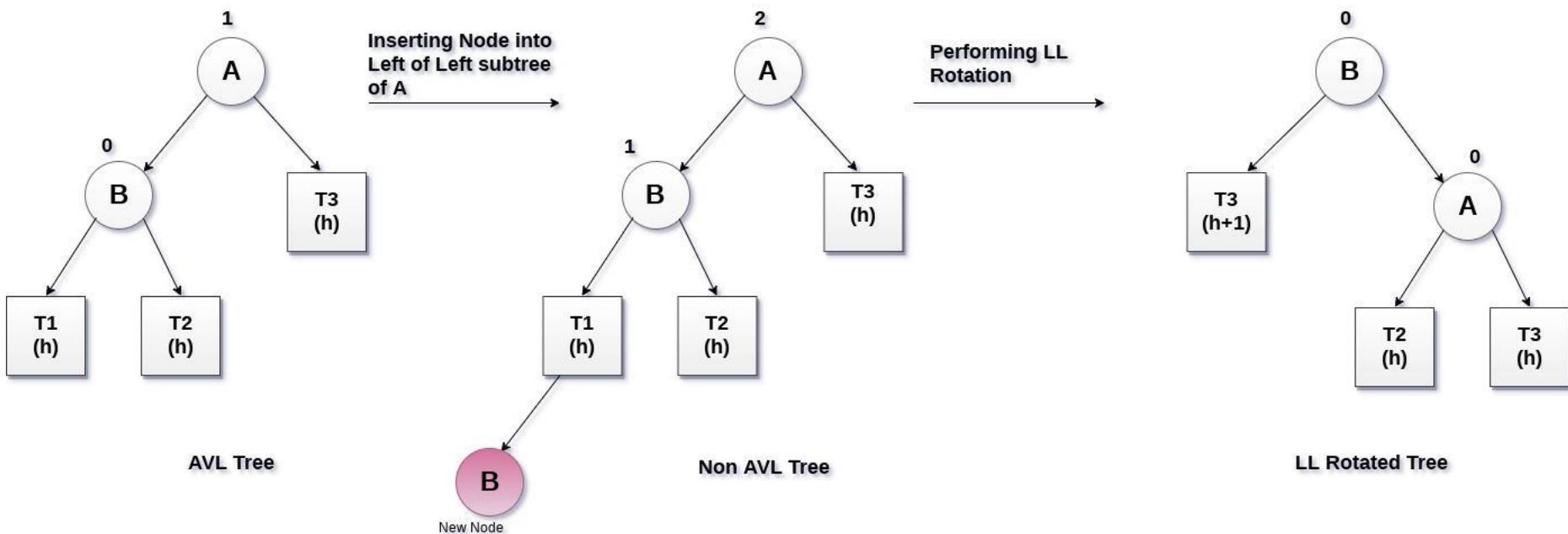


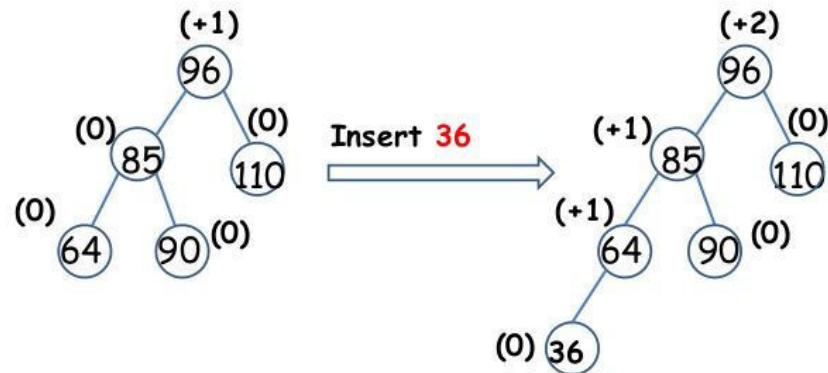
AVL Tree



- **Single Left Rotation (LL Rotation)**

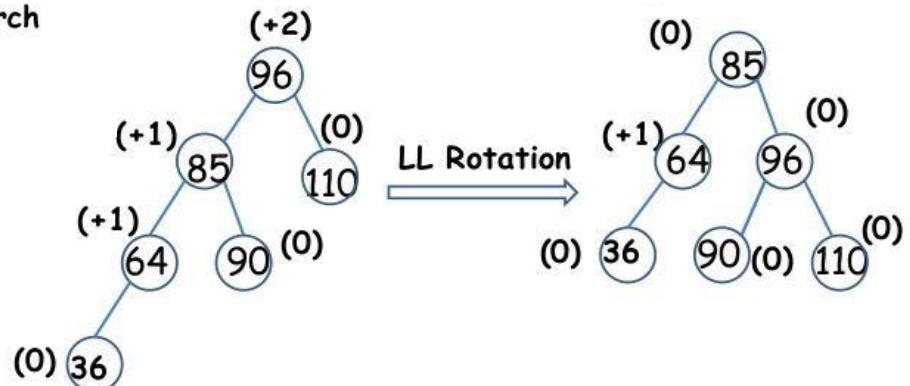
- In LL Rotation every node moves one position to left from the current position. (Note: we see path in which node is inserted/deleted)





Insert 36

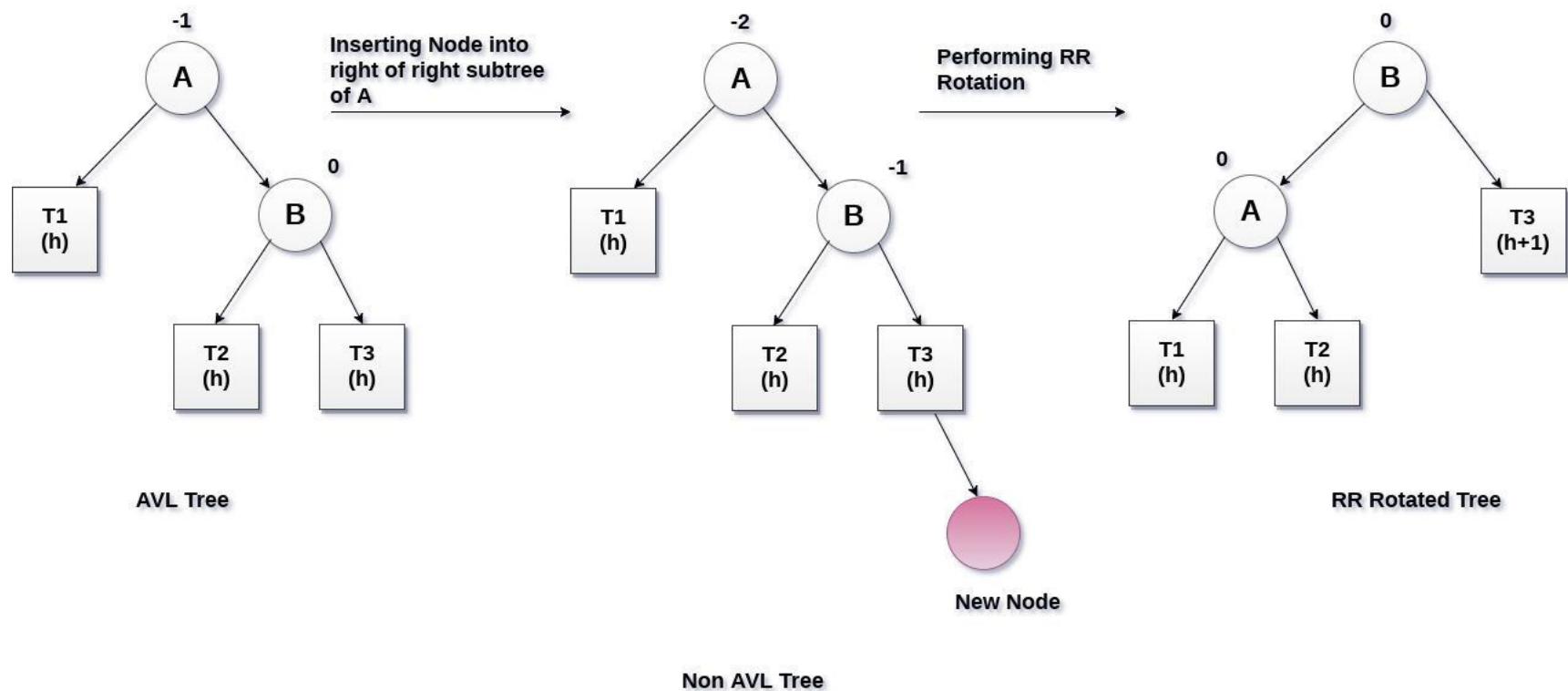
Unbalanced AVL search tree

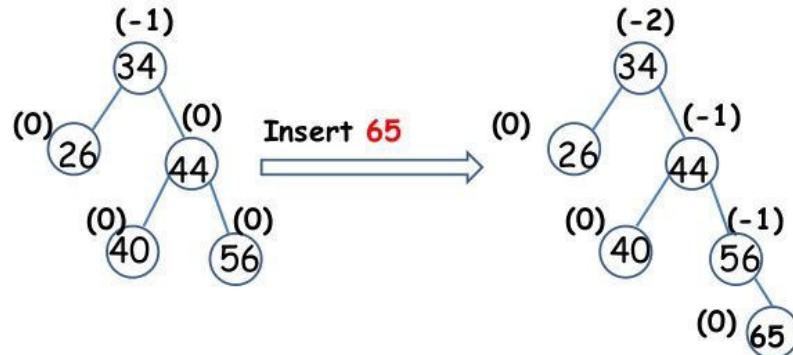


Unbalanced AVL search tree

Balanced AVL search tree
after LL rotation

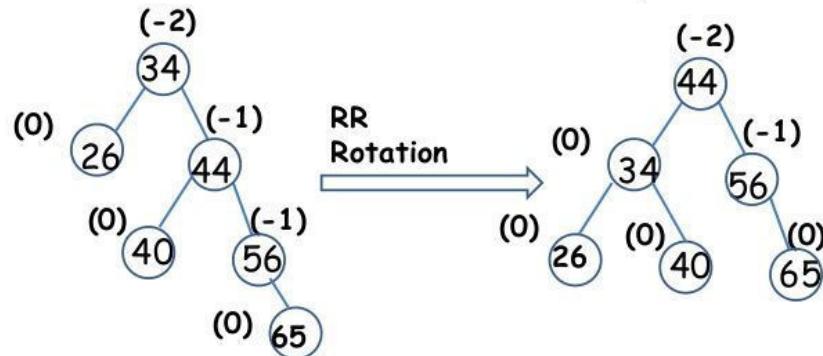
- **Single Right Rotation (RR Rotation)**
 - In RR Rotation every node moves one position to right from the current position.





Insert 65

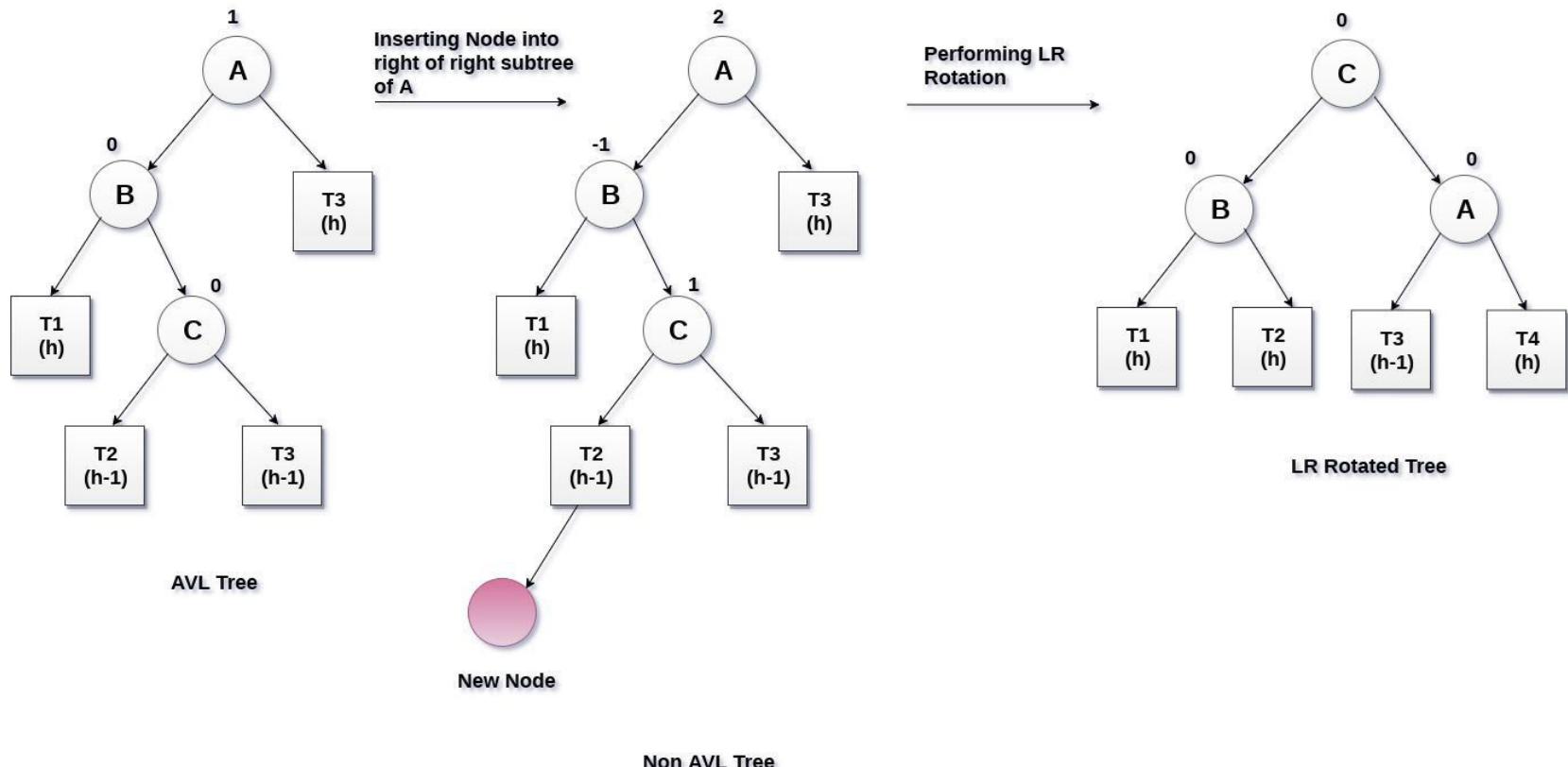
Unbalanced AVL search tree



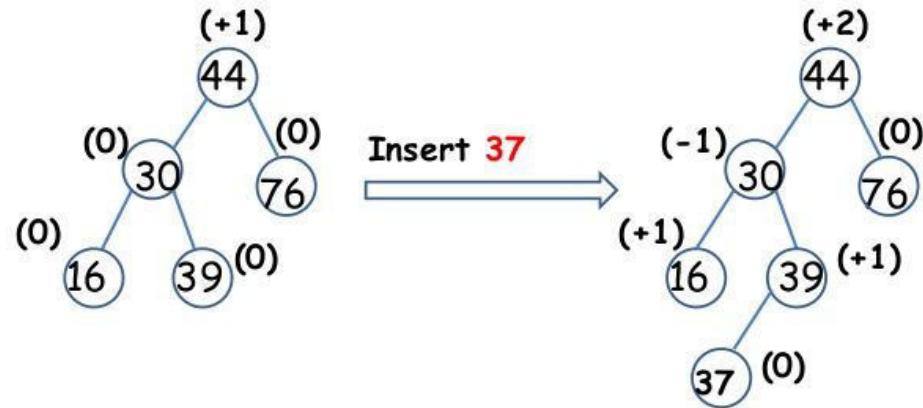
Balanced AVL search tree
after RR rotation

- **Left Right Rotation (LR Rotation)**

- LR rotations is to be performed if the new node is inserted into the right of left sub-tree of the critical node A.
- First single L rotation (RR) and then single R rotation(LL) is performed.

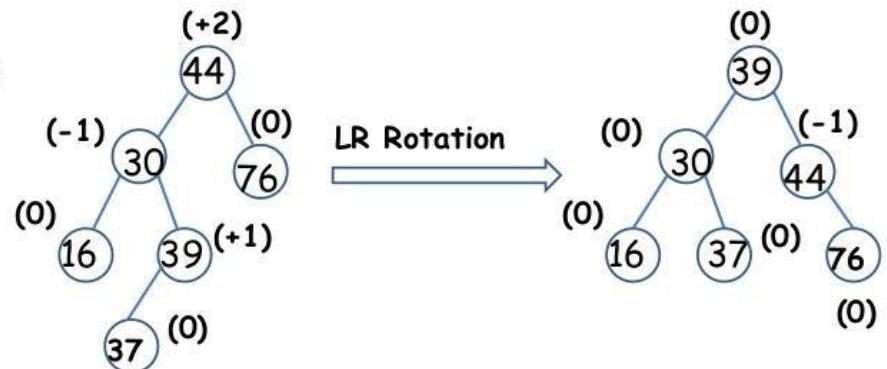


LR Rotation Example



Unbalanced AVL search tree

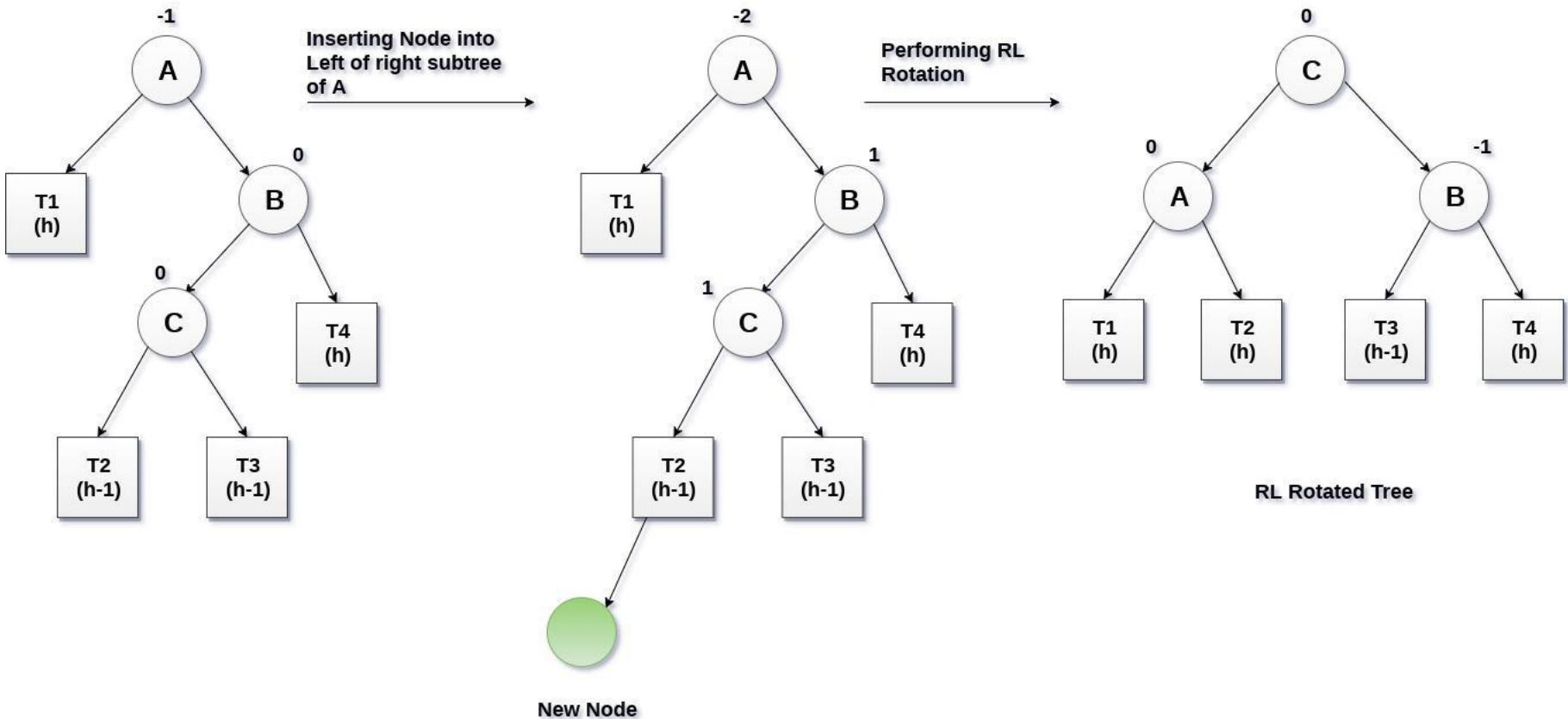
LR Rotation Example



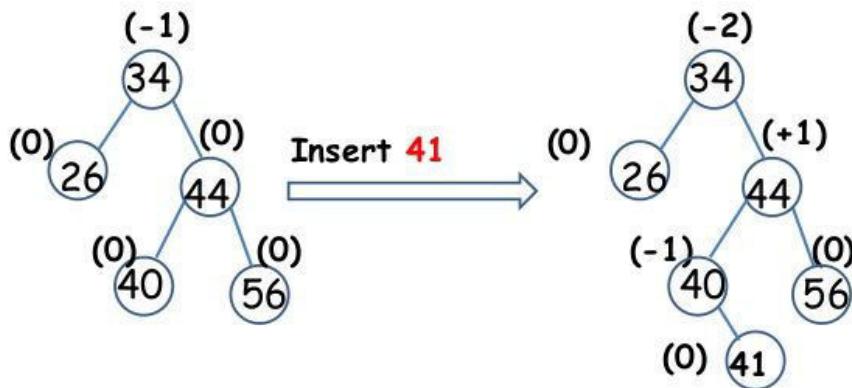
Balanced AVL search tree

• Right Left Rotation (RL Rotation)

- RL rotations is to be performed if the new node is inserted into the left of right sub-tree of the critical node A.
- First single R rotation (LL) and then single L rotation(RR) is performed.

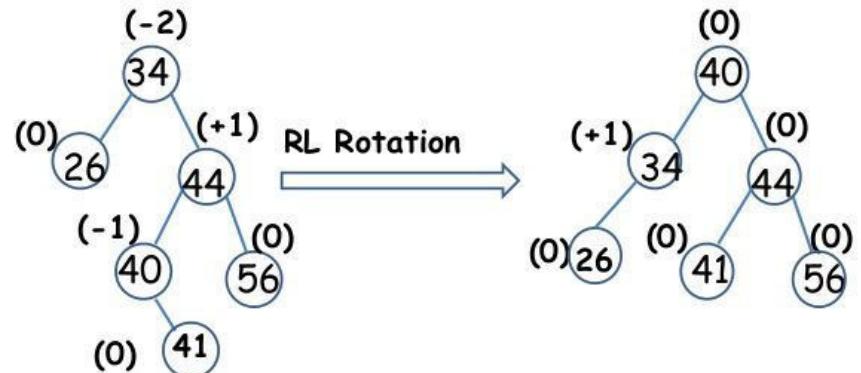


RL Rotation Example



Unbalanced AVL search tree

RL Rotation Example



Balanced AVL search tree

Operations on AVL tree

Algorithm (for insertion):

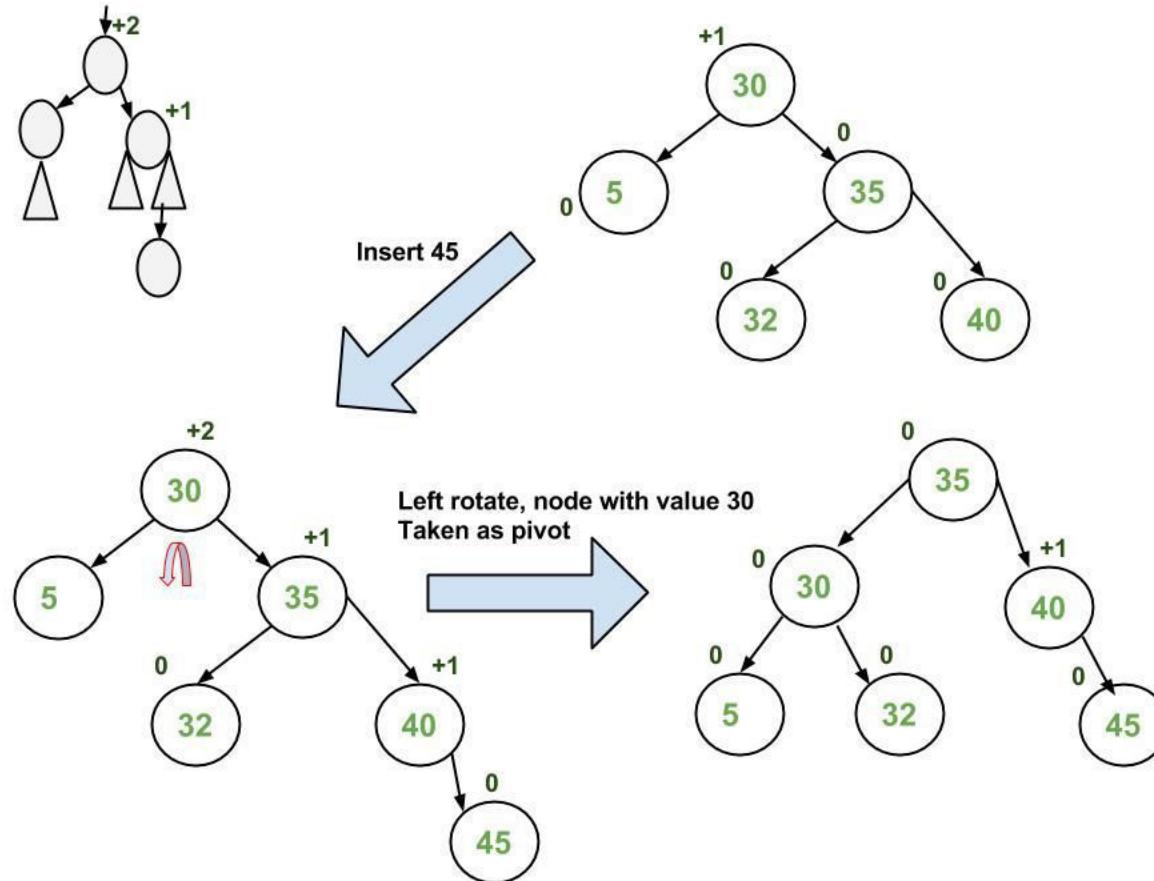
Step 1: insert a new element into the tree using **Binary Search Tree** insertion logic.

Step 2: After inserting the elements, check the **Balance Factor** of each node.

Step 3: When the **Balance Factor** of every node will be like 0 or 1 or -1, algorithm will proceed for the next operation.

Step 4: When the **balance factor** of any node comes other than the above three values, tree is said to be imbalanced.

Then perform the suitable **Rotation** to make it balanced and then the algorithm will proceed for the next operation.

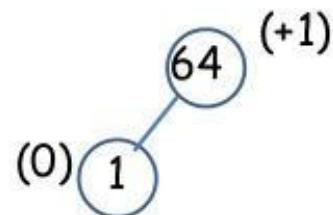


AVL Tree

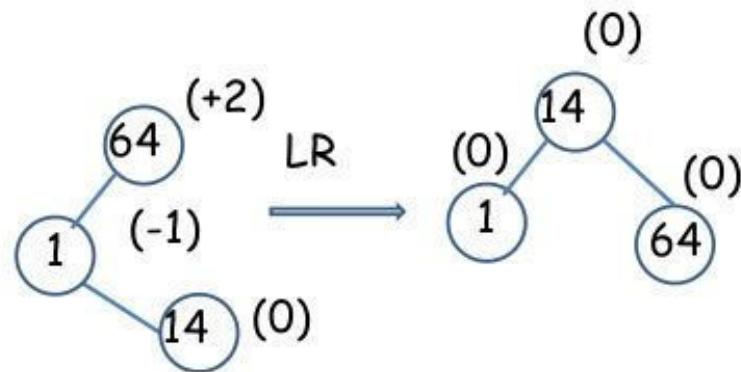
Construct an AVL search tree by inserting the following elements in the order of their occurrence

64, 1, 14, 26, 13, 110, 98, 85

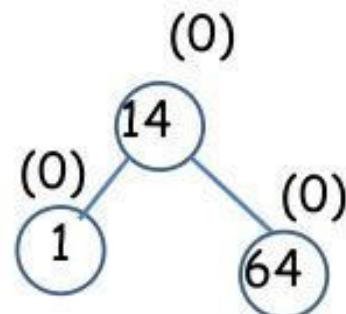
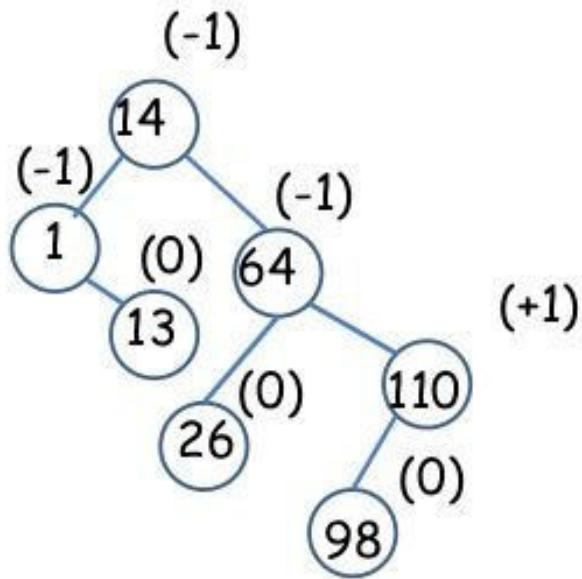
Insert 64, 1



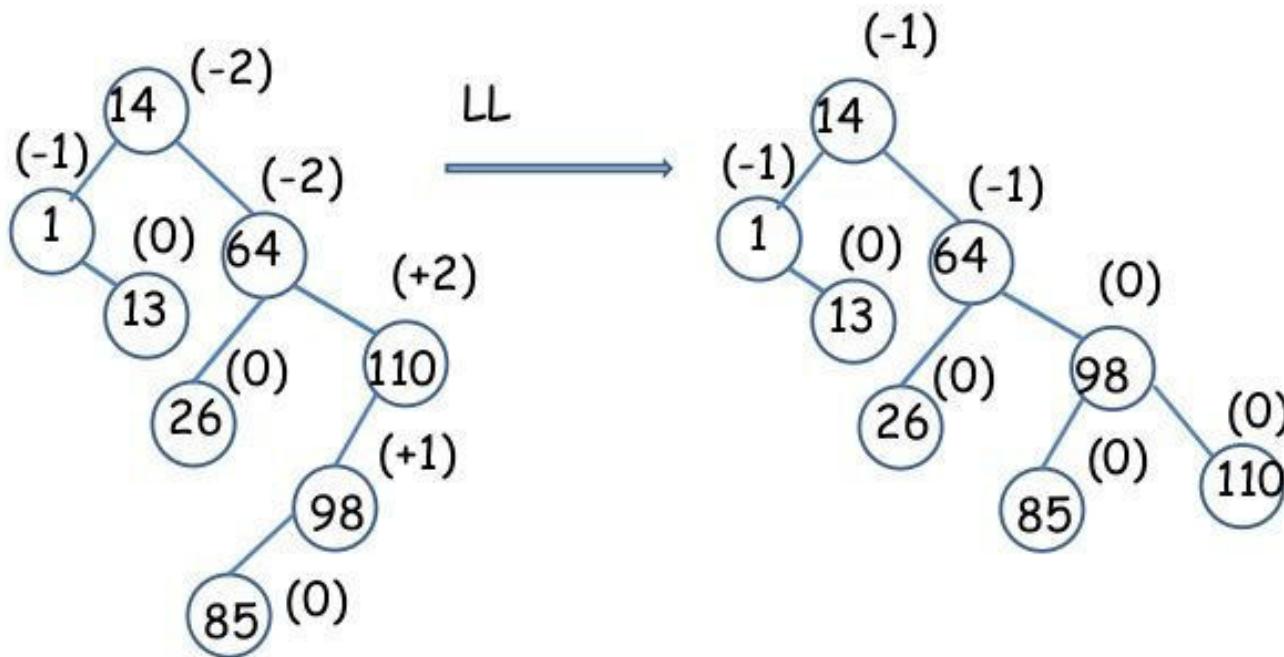
Insert 14



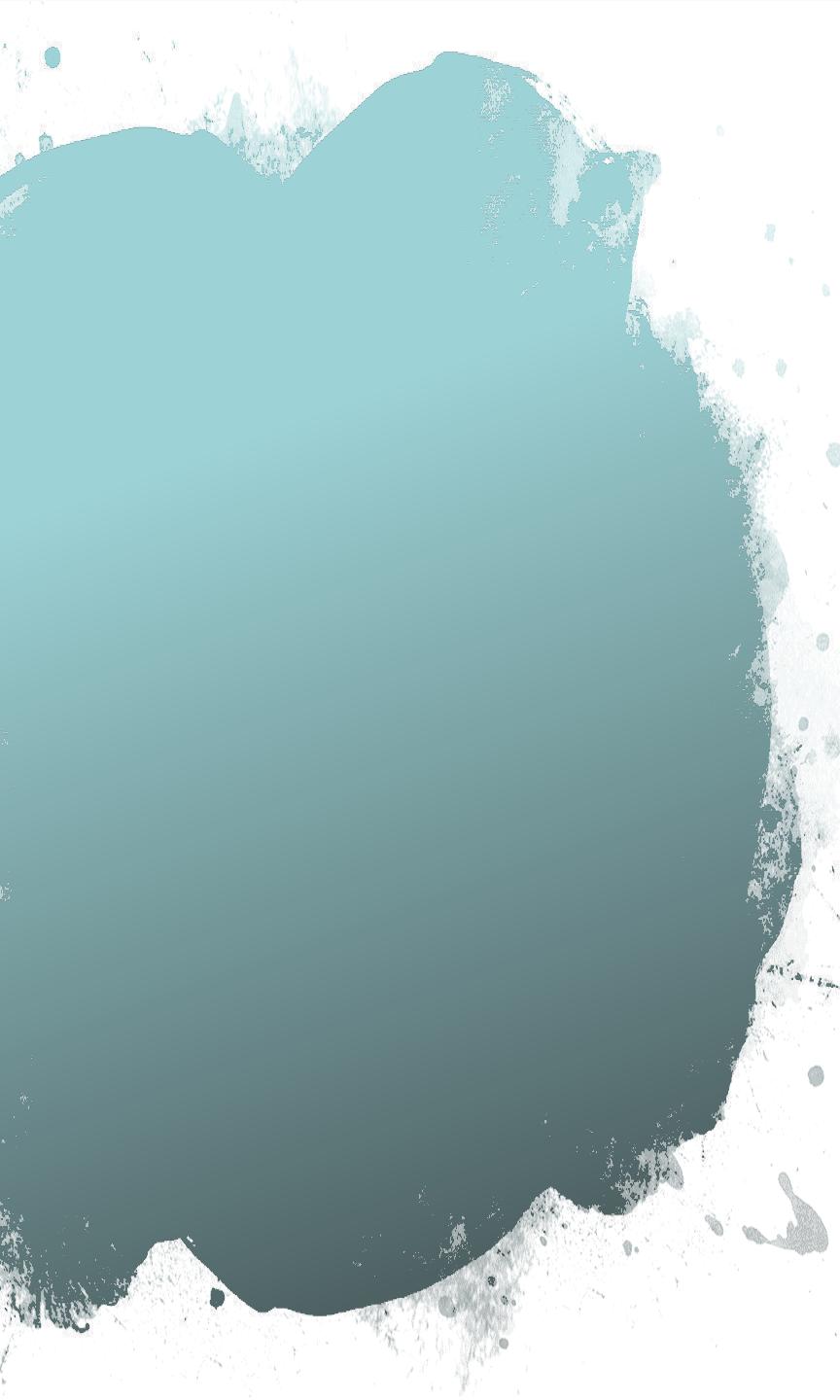
Insert 26, 13, 110, 98



Insert 85







DATA STRUCTURES USING C

Module 4 - Lecture VI

Trees

Prepared By
Ms. Smriti Sehgal

- Deleting a node from an AVL tree is similar to that in a binary search tree.
- Deletion may disturb the balance factor of an AVL tree and therefore the tree needs to be rebalanced in order to maintain the AVLness. To solve this, we need to perform rotations.
- The two types of rotations are:
 - L rotation
 - If the node which is to be deleted is present in the left sub-tree of the critical node, then L rotation is applied
 - R rotation
 - If the node which is to be deleted is present in the right sub-tree of the critical node, the R rotation is applied.

Consider,

A is the critical node

B is the root node of its left sub-tree

If node X, present in the right sub-tree of A, R rotations are performed.

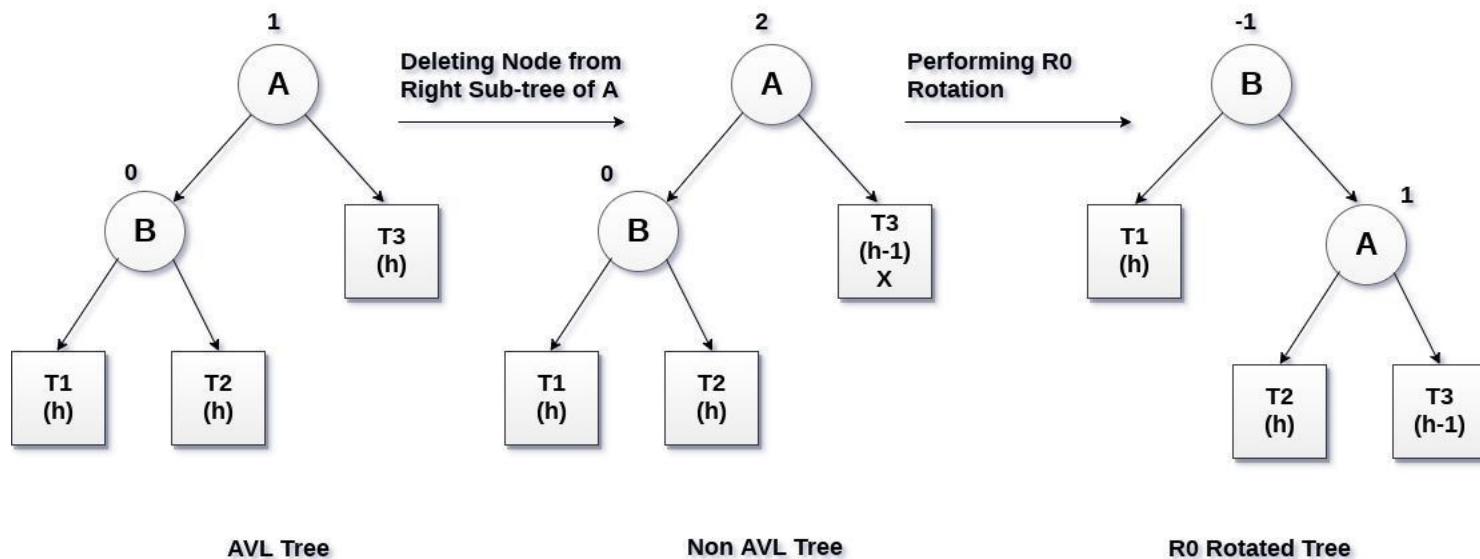
Consider,

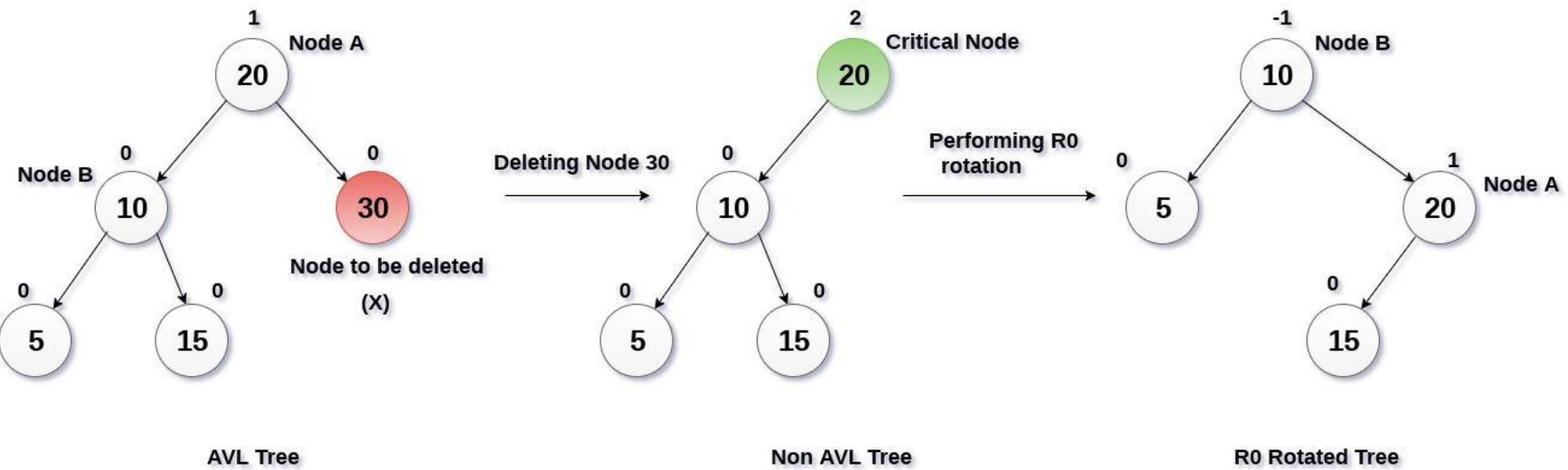
A is the critical node

B is the root node of its right sub-tree

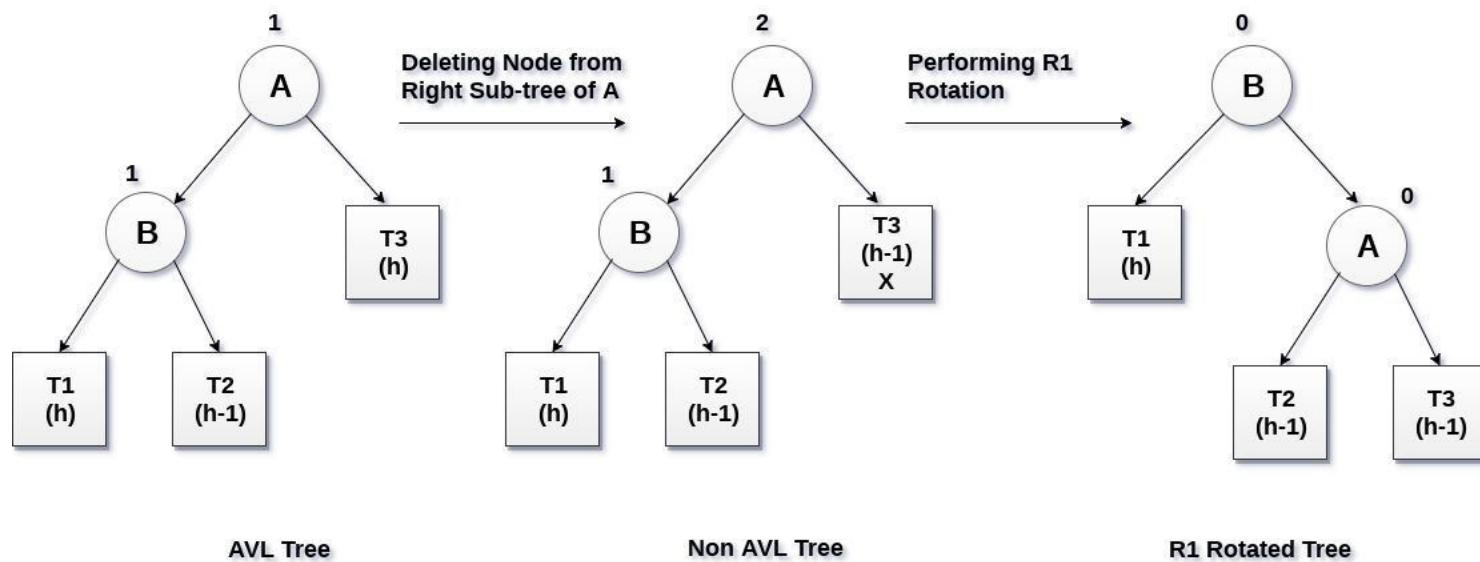
If node X, present in the left sub-tree of A, L rotations are performed.

R0 rotation (Node B has balance factor 0)





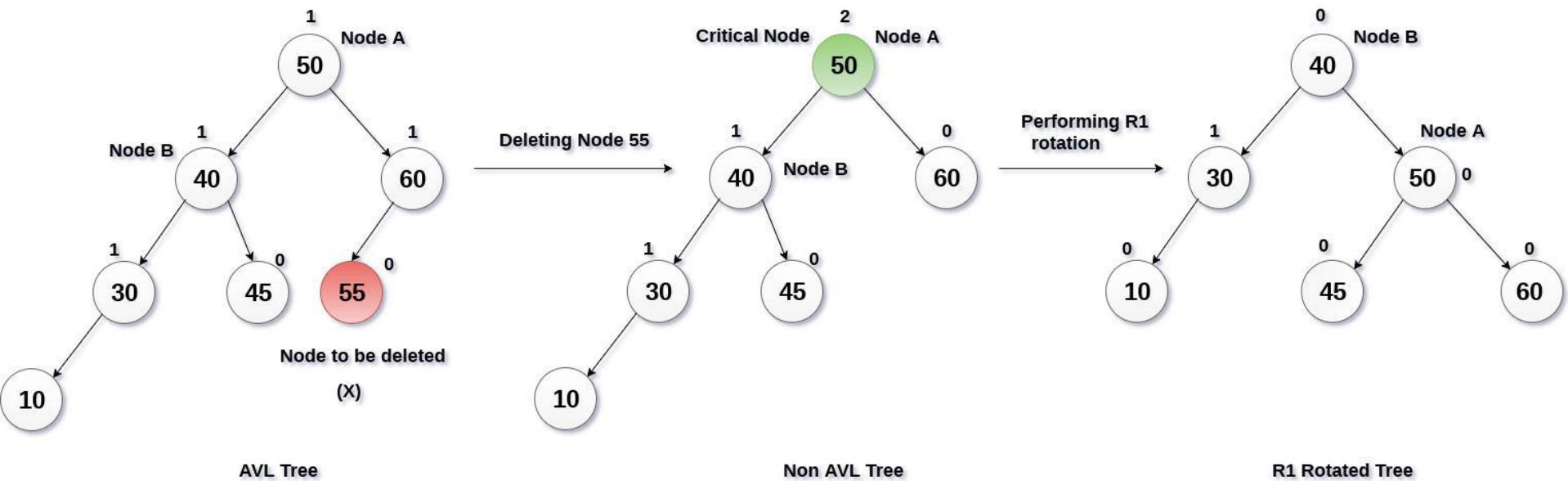
R1 Rotation (Node B has balance factor 1)



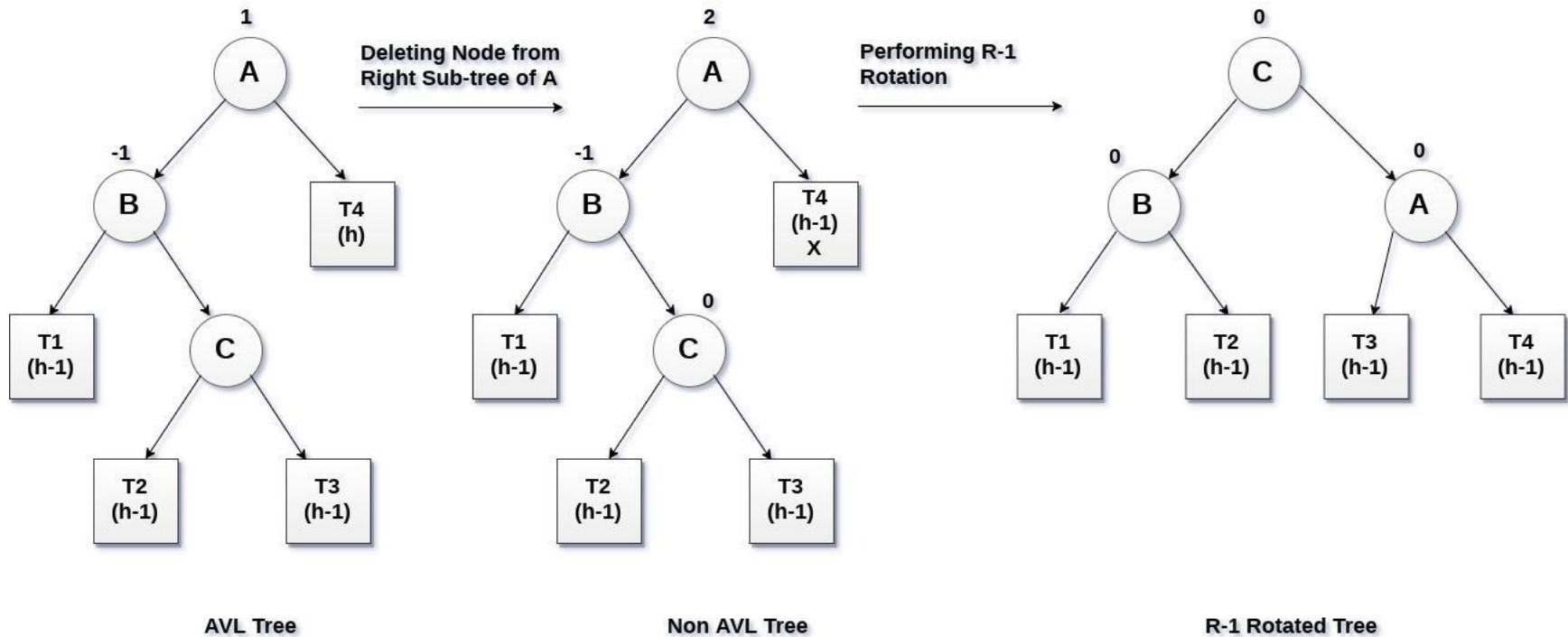
AVL Tree

Non AVL Tree

R1 Rotated Tree



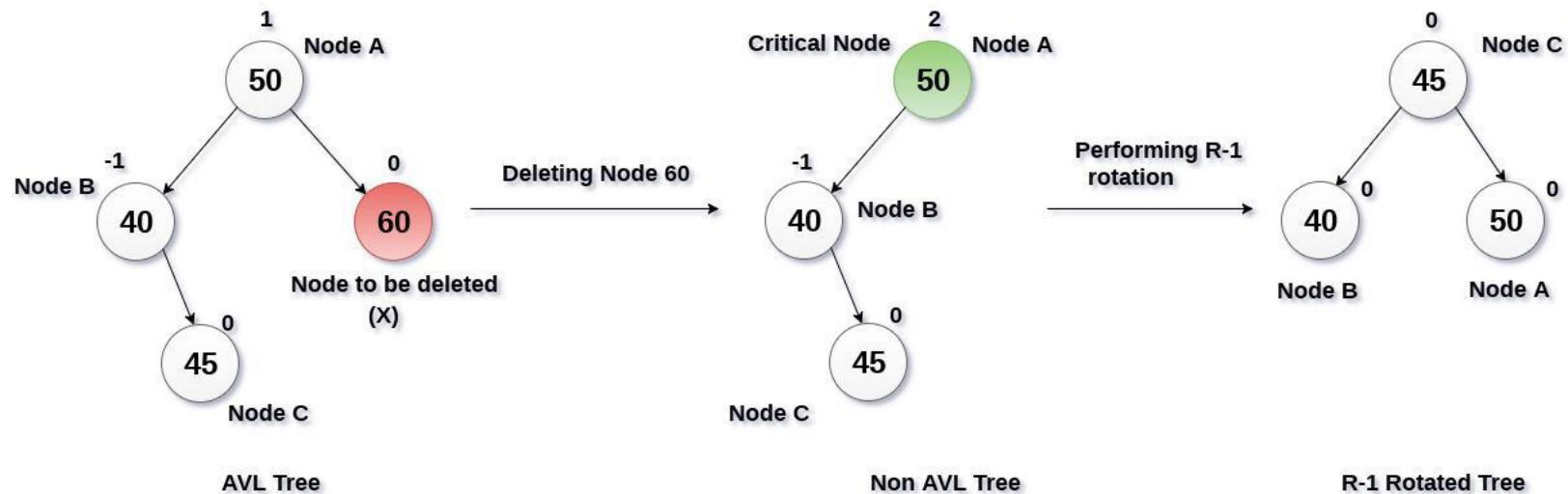
R-1 Rotation (Node B has balance factor -1)



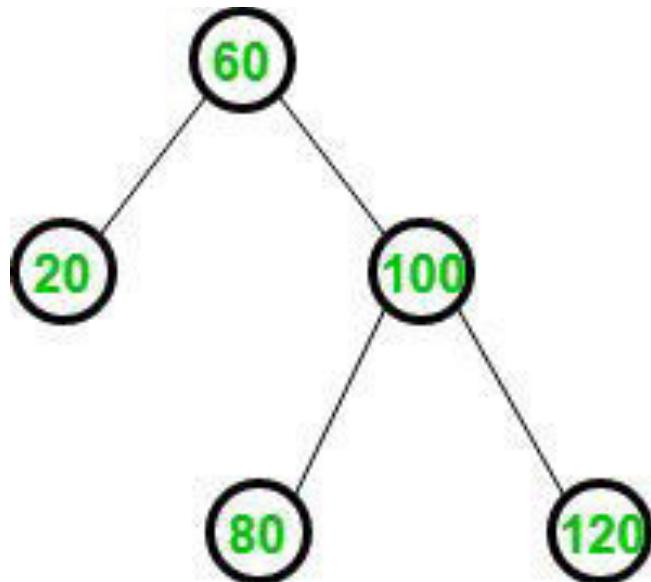
AVL Tree

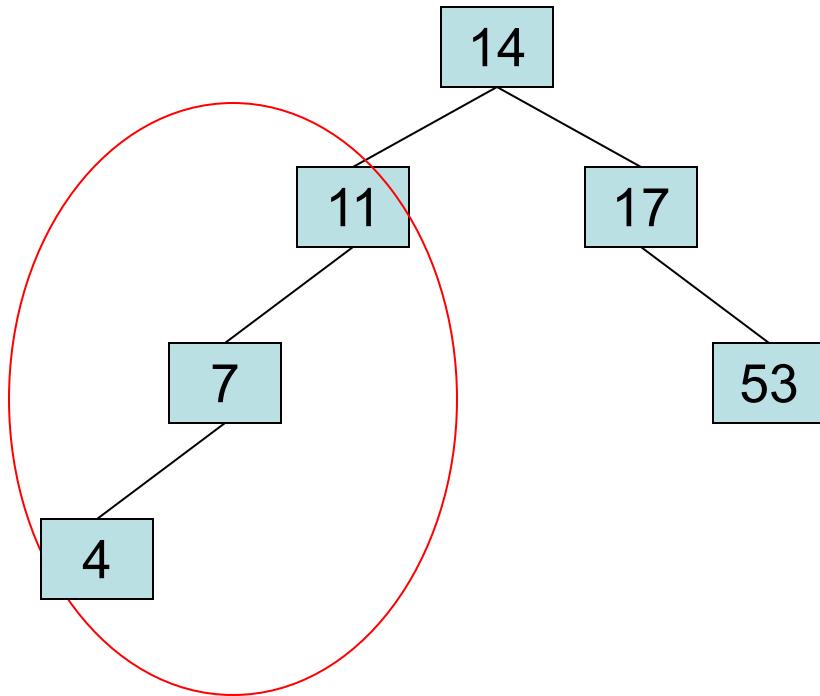
Non AVL Tree

R-1 Rotated Tree



Eg. Add 70

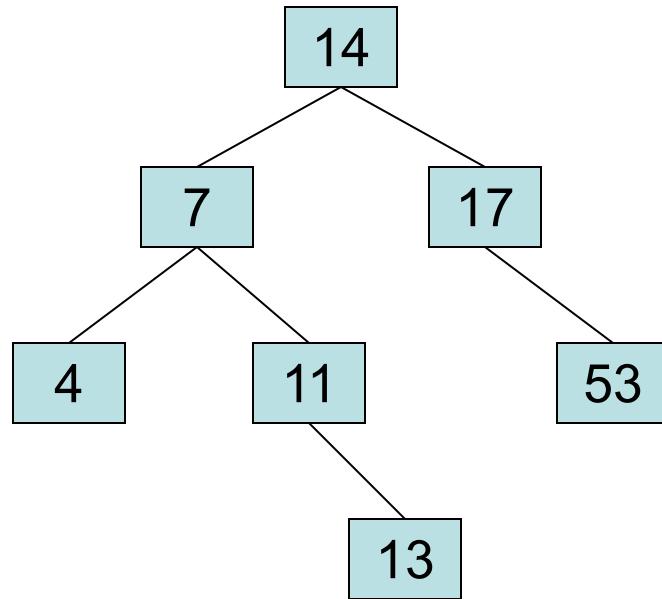


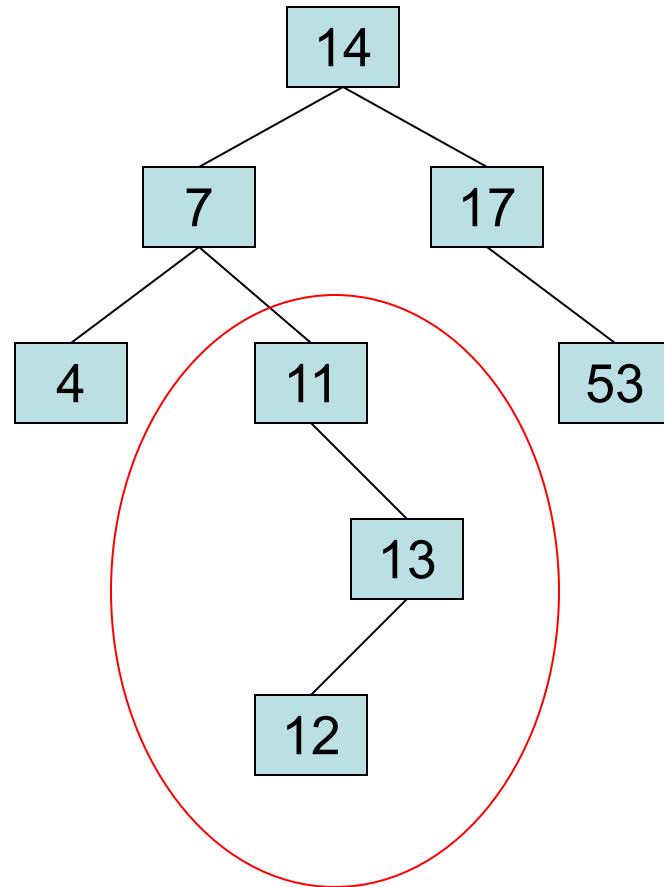


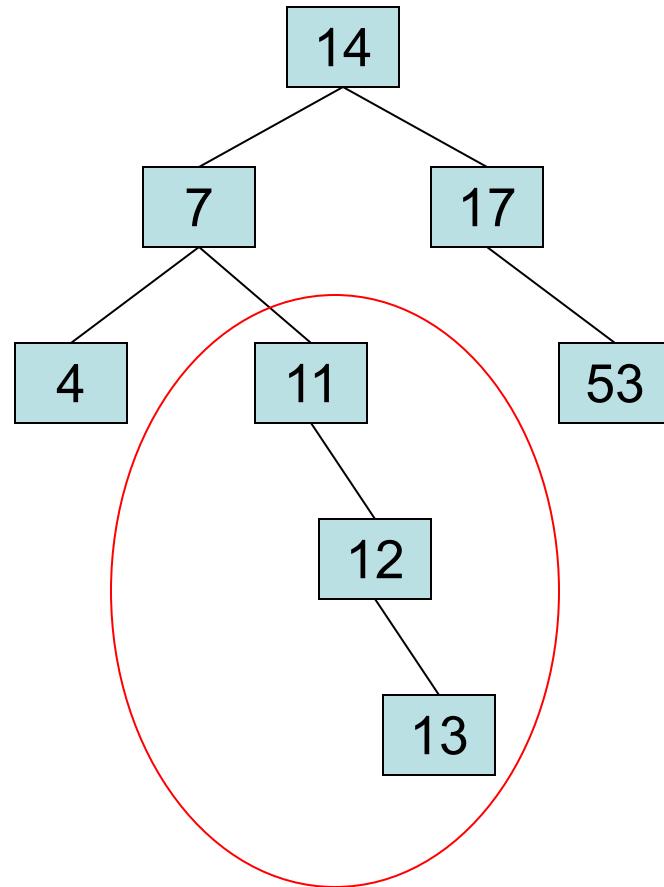
AVL Tree Example:

- Insert 14, 17, 11, 7, 53, 4, 13 into an empty AVL tree

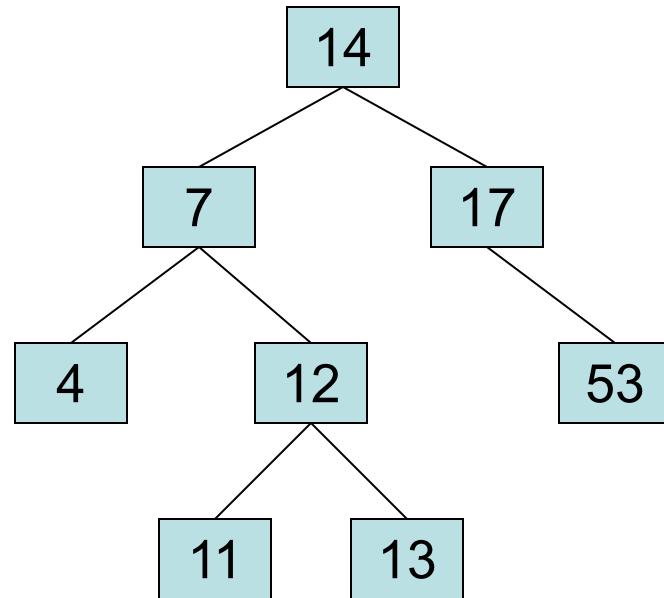
- Insert 14, 17, 11, 7, 53, 4, 13 into an empty AVL tree

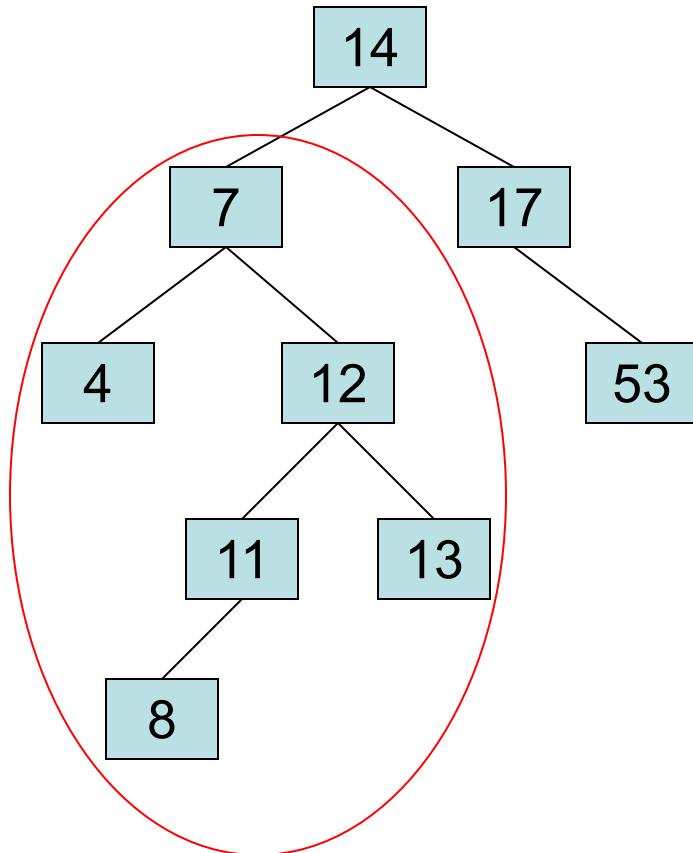


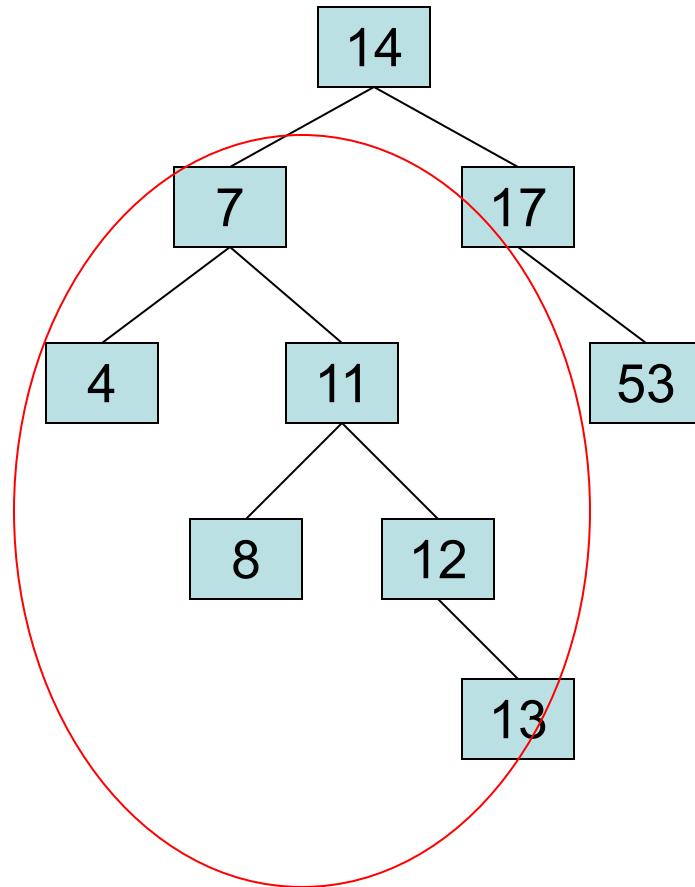




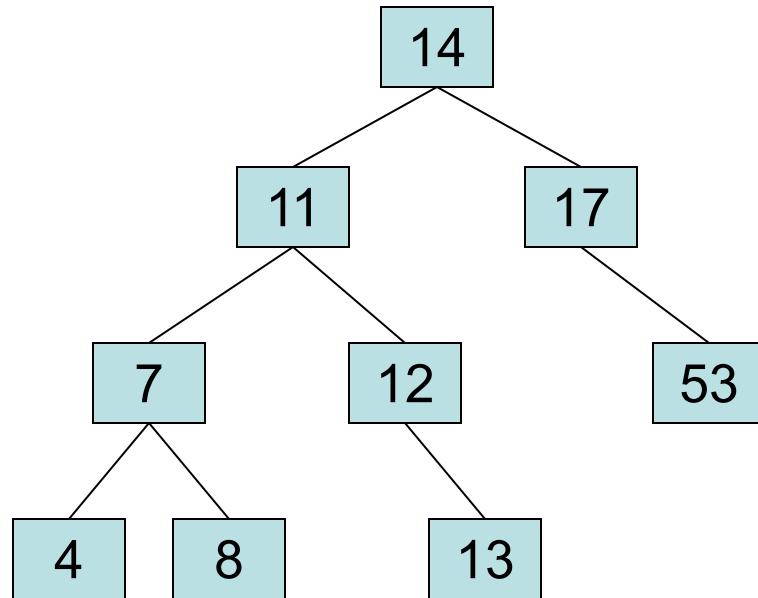
- Now the AVL tree is balanced.

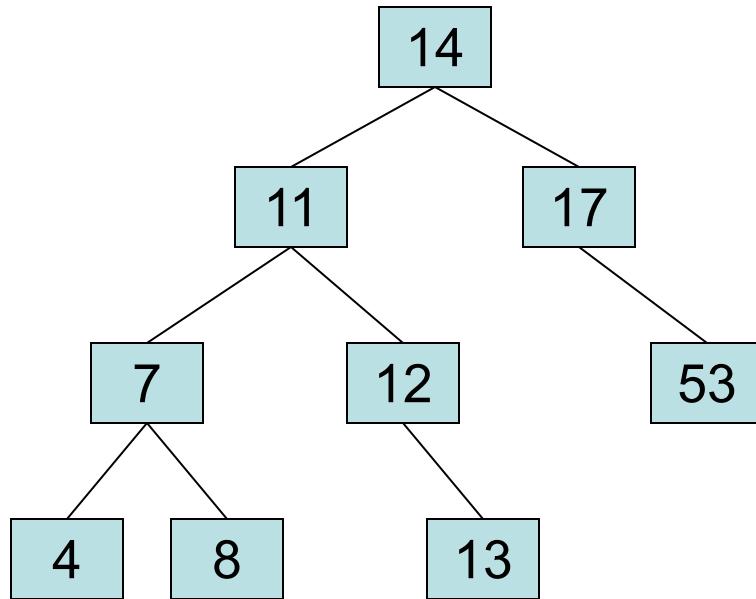






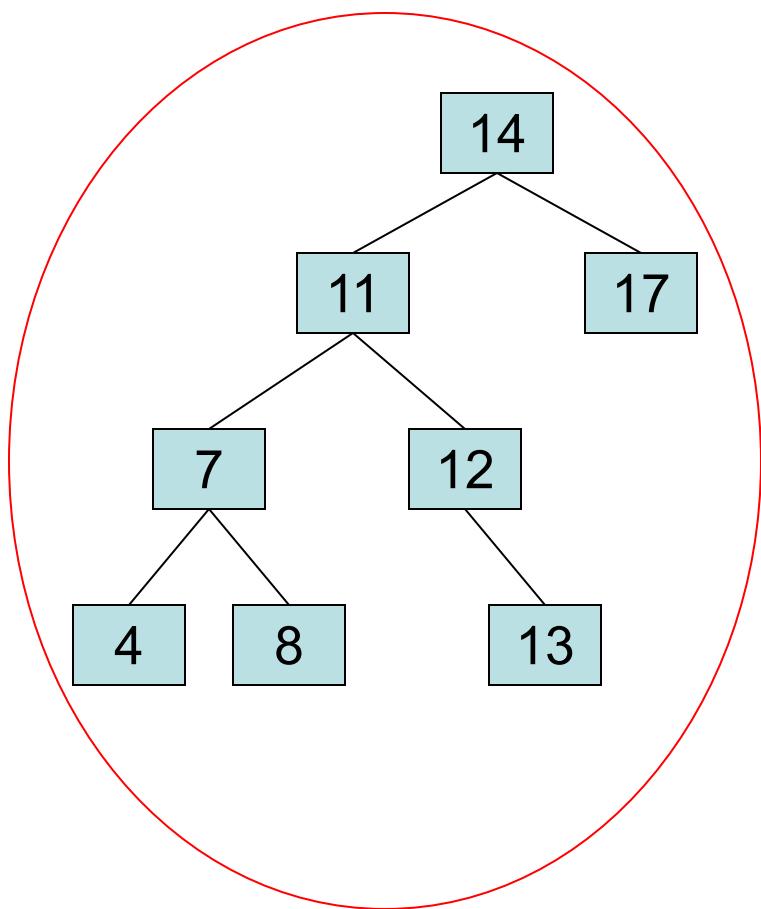
- Now the AVL tree is balanced.





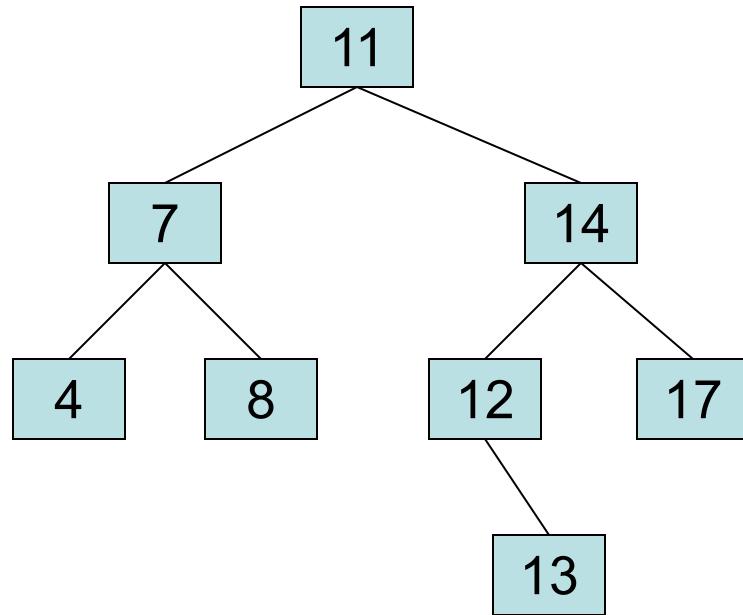
AVL Tree Example:

- Now remove 53



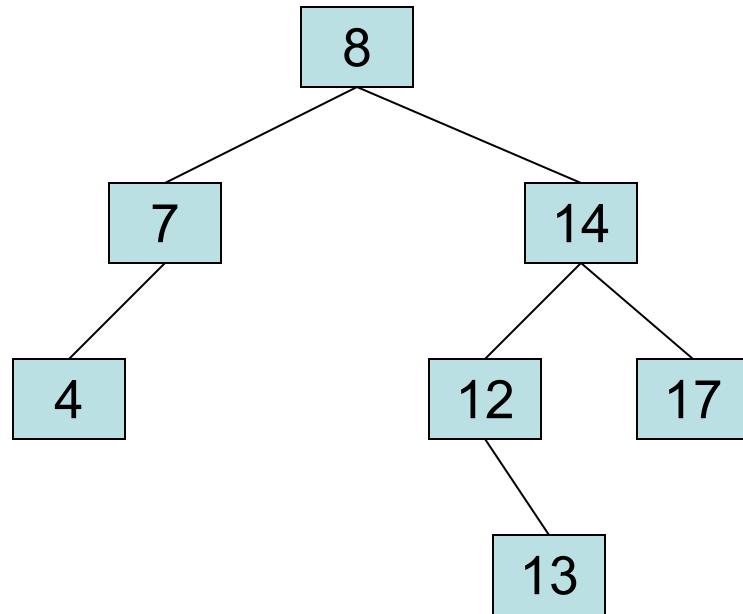
AVL Tree Example:

- Now remove 53, unbalanced



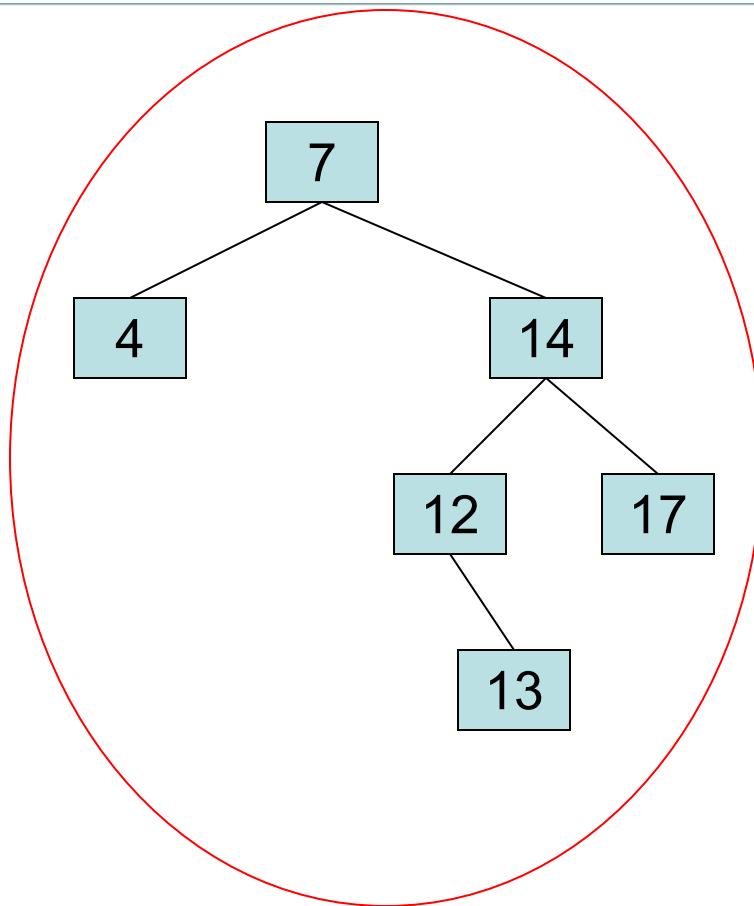
AVL Tree Example:

- **Balanced! Remove 11**



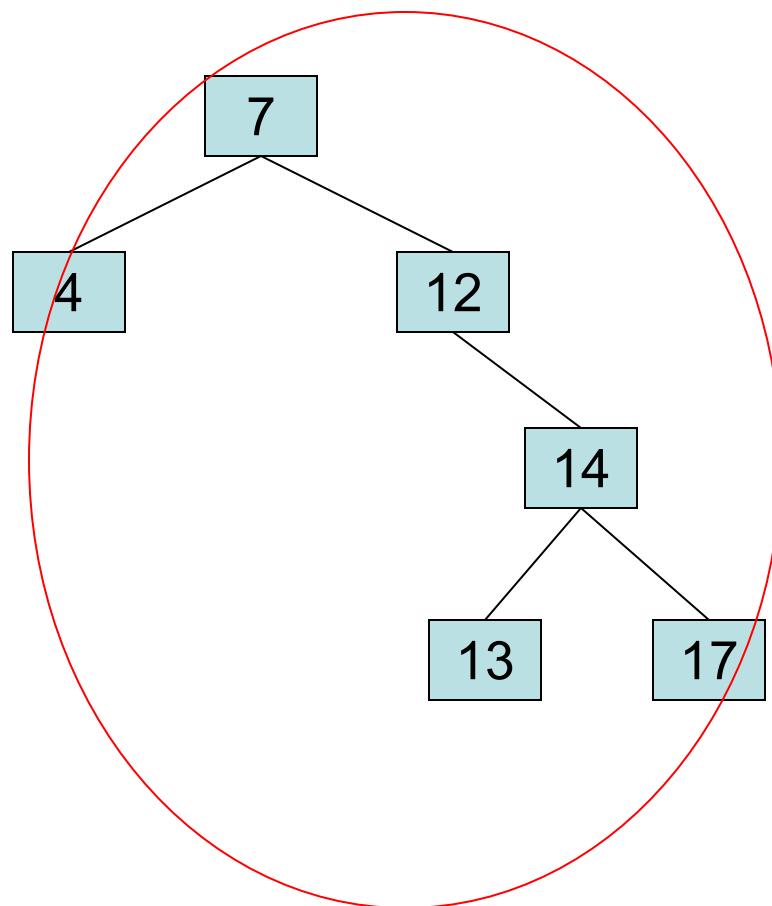
AVL Tree Example:

- Remove 11, replace it with the largest in its left branch



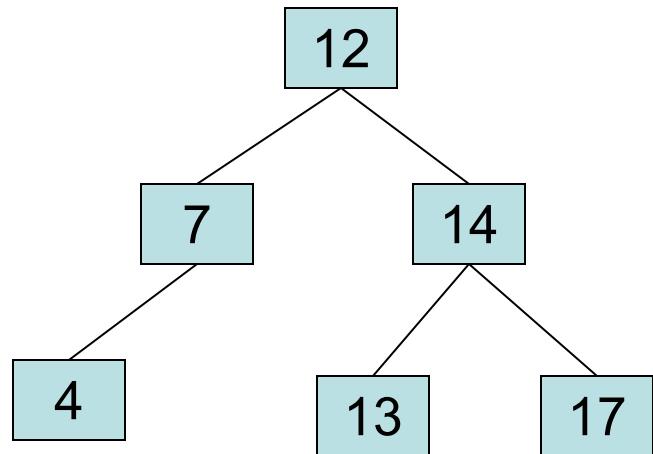
AVL Tree Example:

- Remove 8, unbalanced



AVL Tree Example:

- Remove 8, unbalanced



AVL Tree Example:

- **Balanced!!**

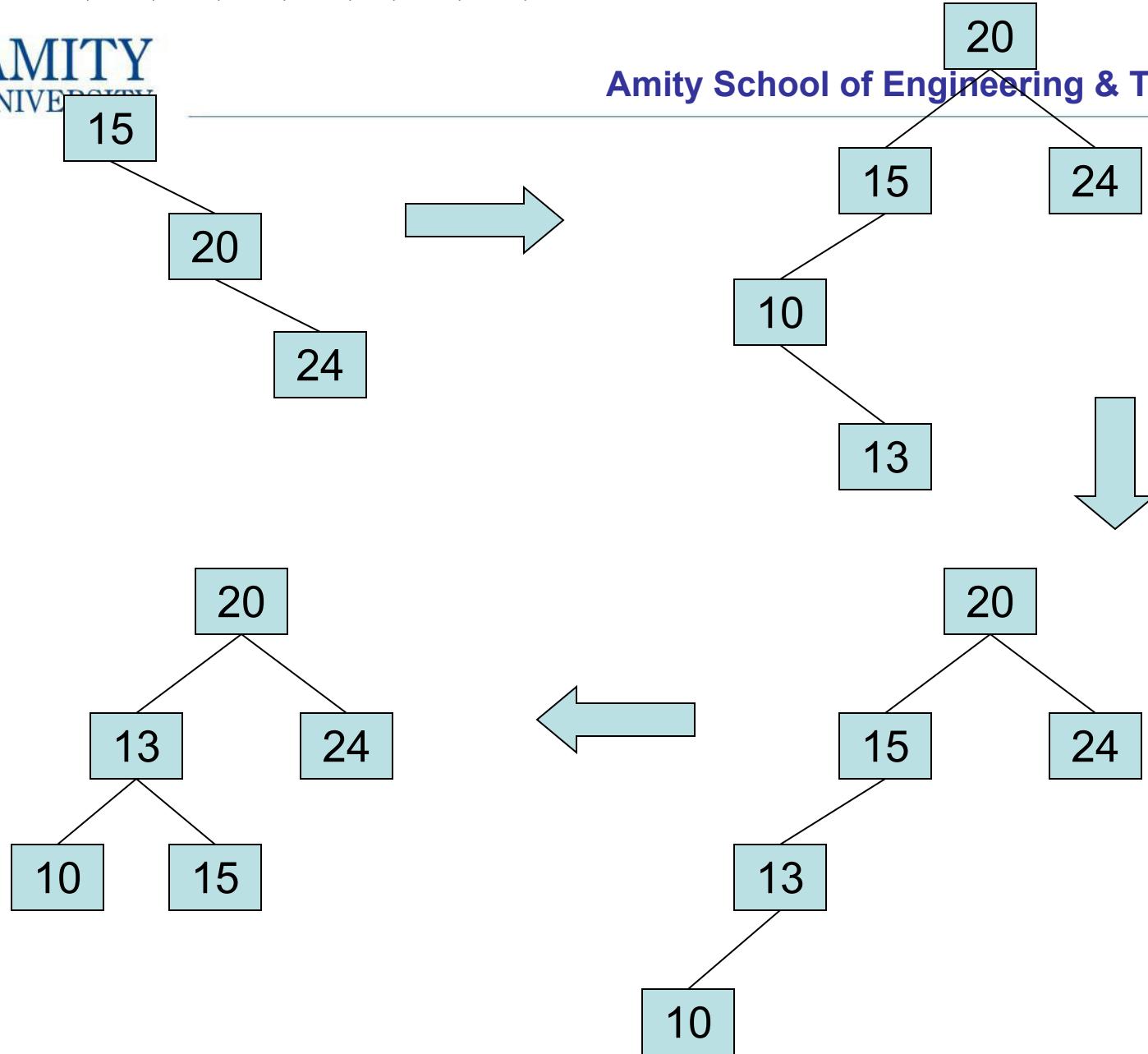
HOME Exercises

- Build an AVL tree with the following values:
15, 20, 24, 10, 13, 7, 30, 36, 25

15, 20, 24, 10, 13, 7, 30, 36, 25



Amity School of Engineering & Technology

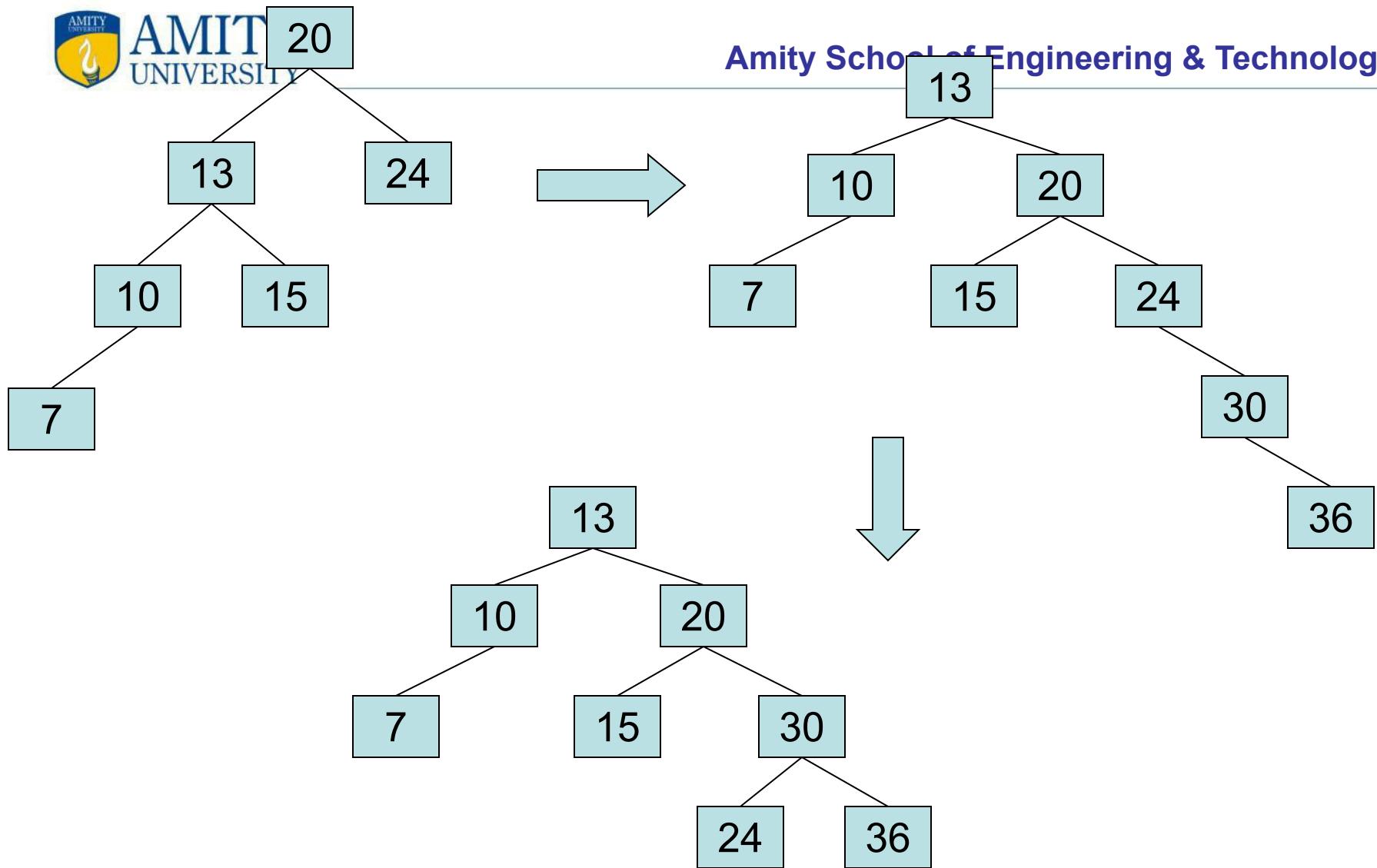


15, 20, 24, 10, 13, 7, 30, 36, 25



AMITY
UNIVERSITY

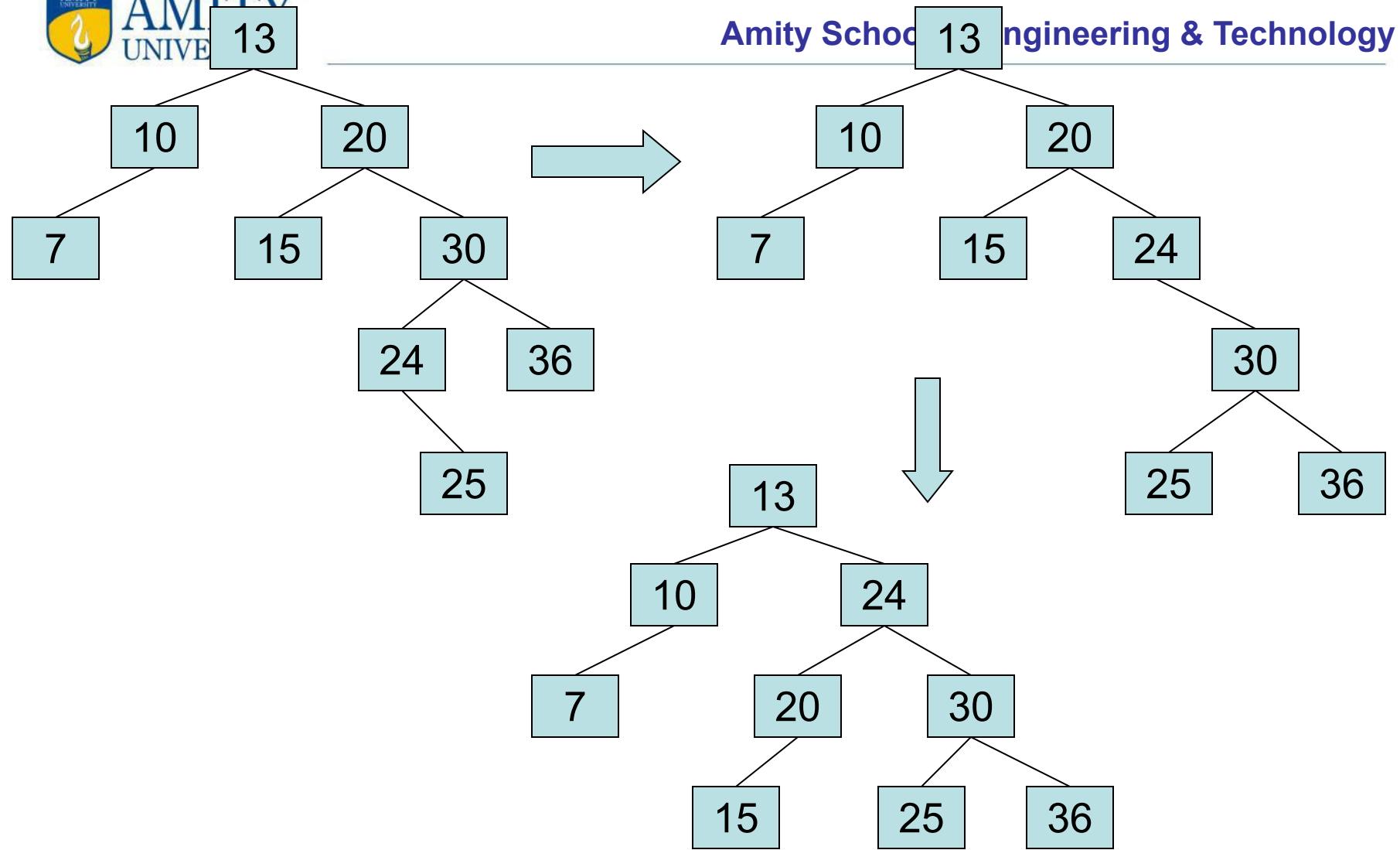
Amity School of Engineering & Technology



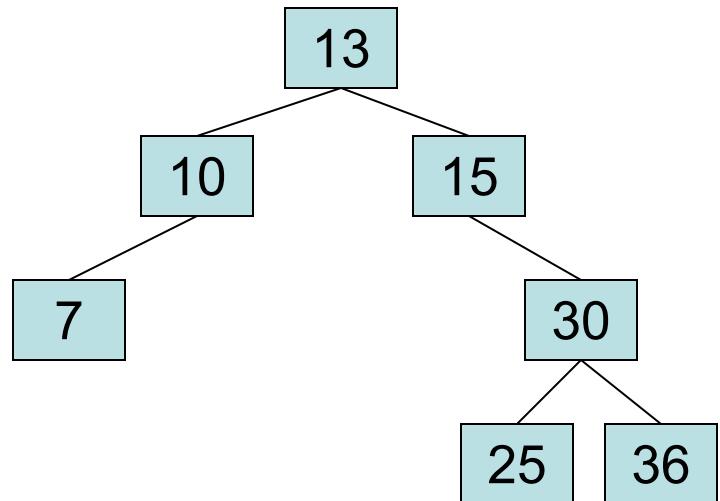
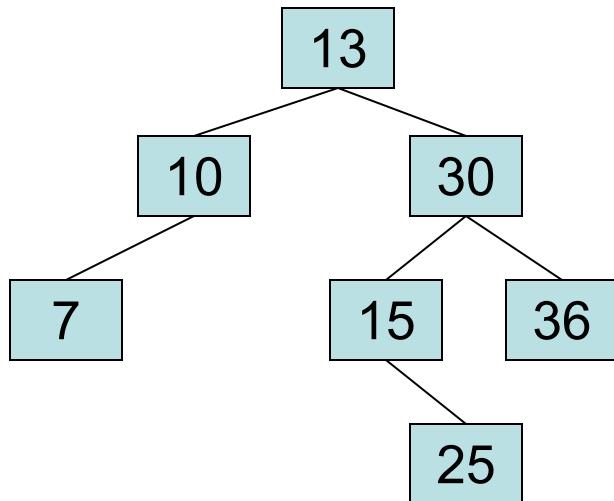
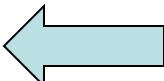
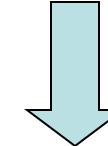
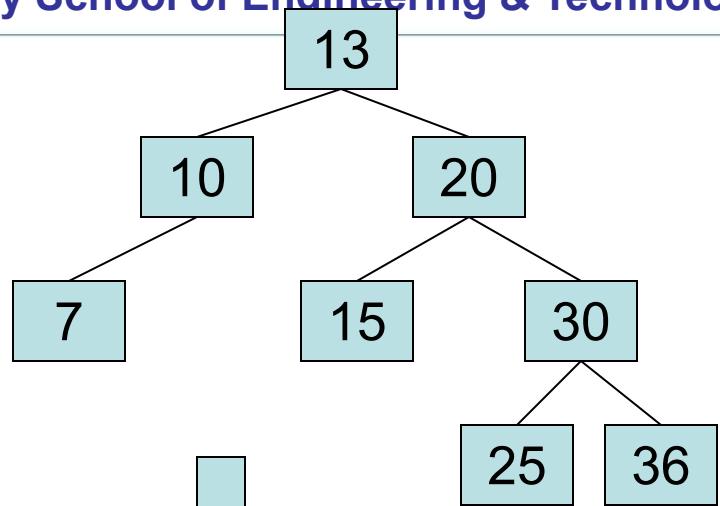
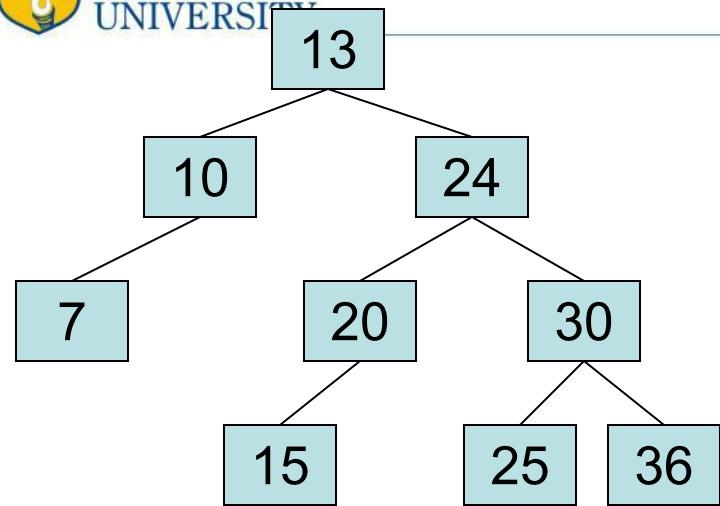
15, 20, 24, 10, 13, 7, 30, 36, 25



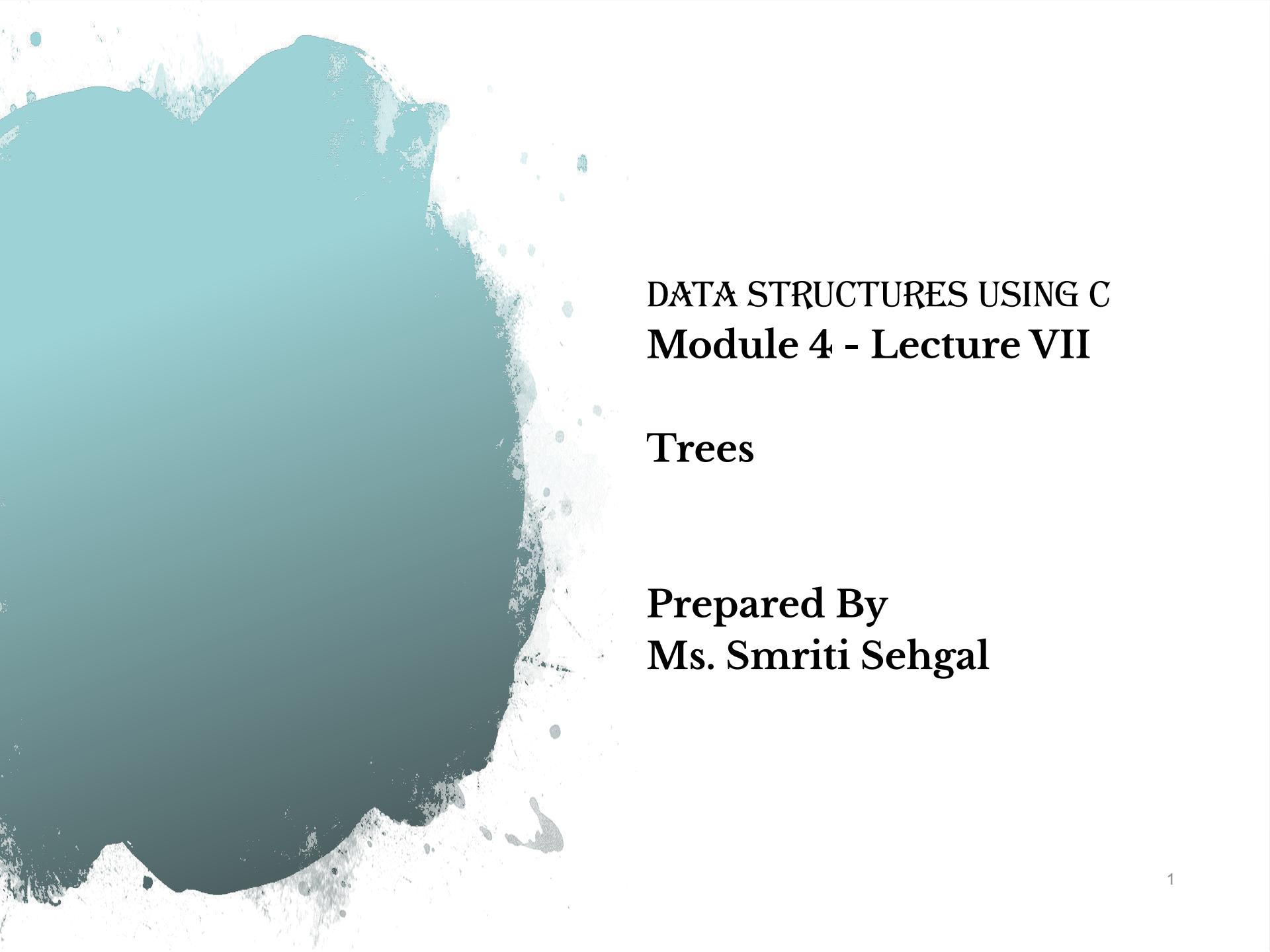
AMITY
UNIVERSITY



Remove 24 and 20 from the AVL tree.







DATA STRUCTURES USING C

Module 4 - Lecture VII

Trees

Prepared By
Ms. Smriti Sehgal

B-Trees

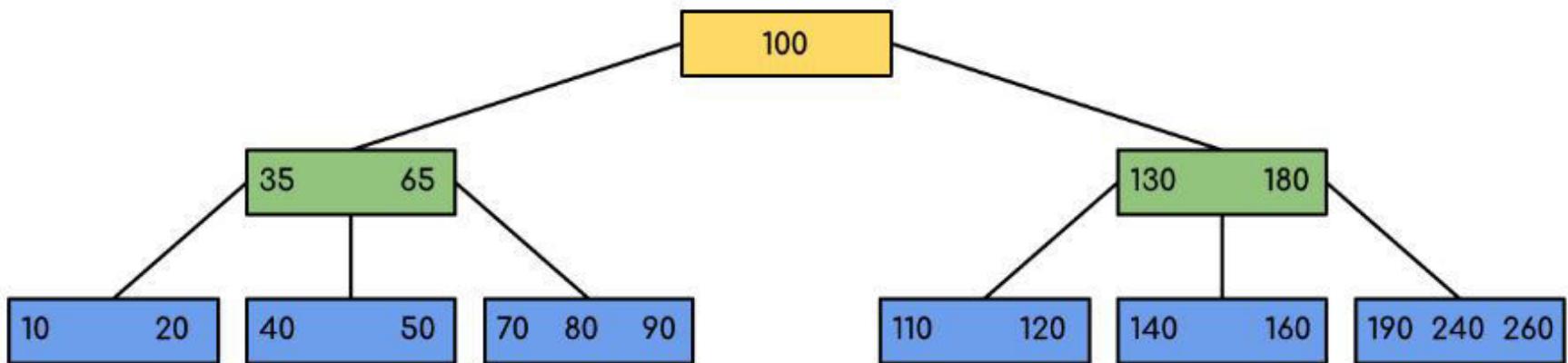
- B-Tree is a self-balancing search tree.
- B-Tree is a tree in which a node contains more than one value (key) and more than two children.
- The number of keys in a node and number of children for a node depends on the order of B-Tree. Every B-Tree has an order.

“m” is the total number of elements in the B-tree.

Properties of B-Tree:

- **Property #1** - All **leaf nodes** must be **at same level**.
- **Property #2** - All nodes except root must have at least $[m/2]-1$ keys and maximum of **m-1** keys.
- **Property #3** - All non leaf nodes except root (i.e. all internal nodes) must have at least **$m/2$** children.
- **Property #4** - If the root node is a non leaf node, then it must have **atleast 2** children.
- **Property #5** - A non leaf node with **n-1** keys must have **n** number of children.
- **Property #6** - All the **key values in a node** must be in **Ascending Order**.

B-Tree with order 5



(non-root)	Min	max	Eg.	Min	max
Keys	$\text{ceil}(m/2)-1$	$m-1$	Keys	2	4
Children	$\text{Ceil}(m/2)$	m	Children	3	5

Operations on B-Tree

- Searching
- Insertion
- Deletion



Step 1 - Read the search element from the user.

Step 2 - Compare the search element with first key value of root node in the tree.

Step 3 - If both are matched, then display "Given node is found!!!" and terminate the function

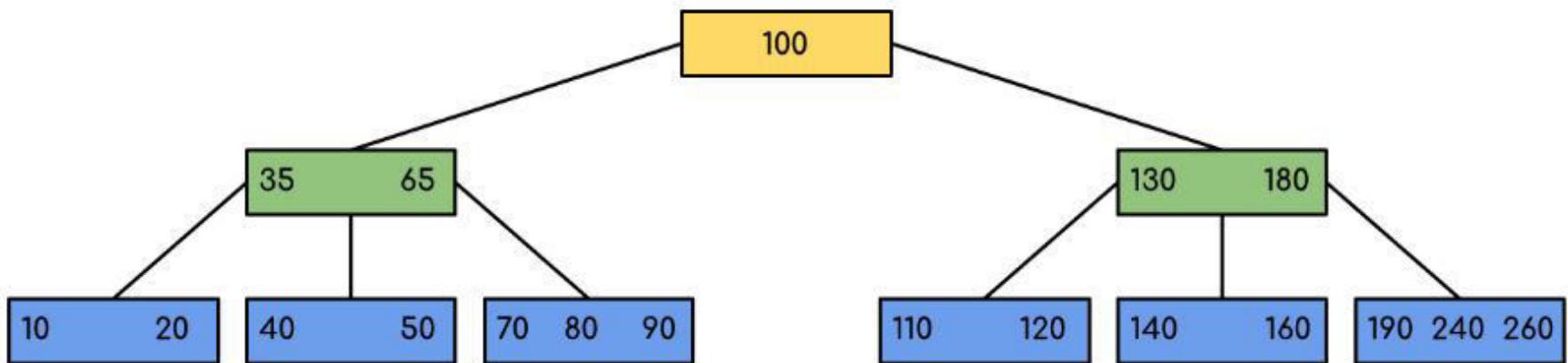
Step 4 - If both are not matched, then check whether search element is smaller or larger than that key value.

Step 5 - If search element is smaller, then continue the search process in left subtree.

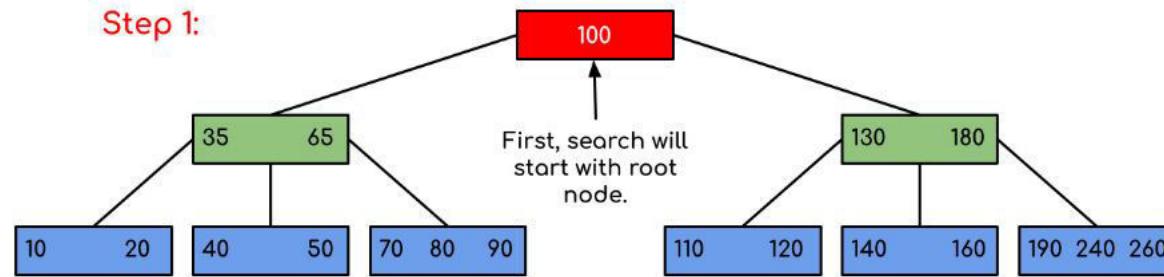
Step 6 - If search element is larger, then compare the search element with next key value in the same node and repeat steps 3, 4, 5 and 6 until we find the exact match or until the search element is compared with last key value in the leaf node.

Step 7 - If the last key value in the leaf node is also not matched then display "Element is not found" and terminate the function.

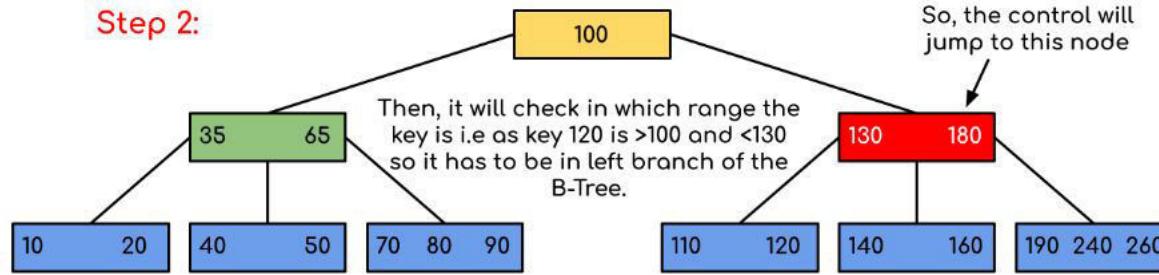
Searching 120 in the given B-Tree



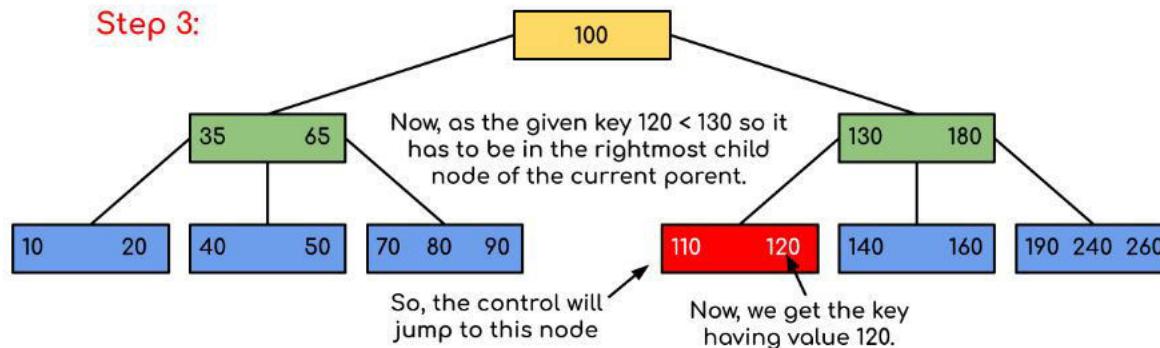
Step 1:



Step 2:



Step 3:



Step 1 - Check whether tree is Empty.

Step 2 - If tree is **Empty**, then create a new node with new key value and insert it into the tree as a root node.

Step 3 - If tree is **Not Empty**, then find the suitable leaf node to which the new key value is added using Binary Search Tree logic.

Step 4 - If that leaf node has empty position, add the new key value to that leaf node in ascending order of key value within the node.

Step 5 - If that leaf node is already full, **split** that leaf node by sending middle value to its parent node. Repeat the same until the sending value is fixed into a node.

Step 6 - If the splitting is performed at root node then the middle value becomes new root node for the tree and the height of the tree is increased by one.

Construct a B-Tree of order 3 by inserting numbers from 1 to 10.

insert(1)

Since '1' is the first element into the tree that is inserted into a new node. It acts as the root node.



insert(2)

Element '2' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node has an empty position. So, new element (2) can be inserted at that empty position.



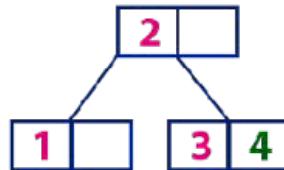
insert(3)

Element '3' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node doesn't have an empty position. So, we split that node by sending middle value (2) to its parent node. But here, this node doesn't have parent. So, this middle value becomes a new root node for the tree.

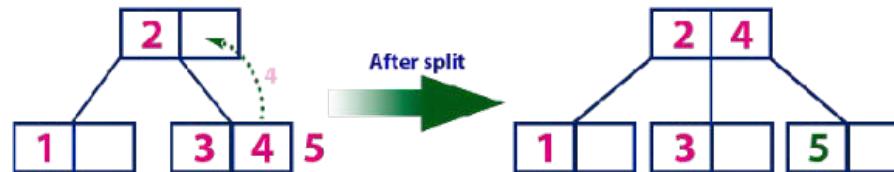


insert(4)

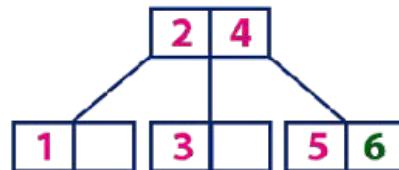
Element '4' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node with value '3' and it has an empty position. So, new element (4) can be inserted at that empty position.


Insert(5)

Element '5' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (4) to its parent node (2). There is an empty position in its parent node. So, value '4' is added to node with value '2' and new element '5' added as new leaf node.

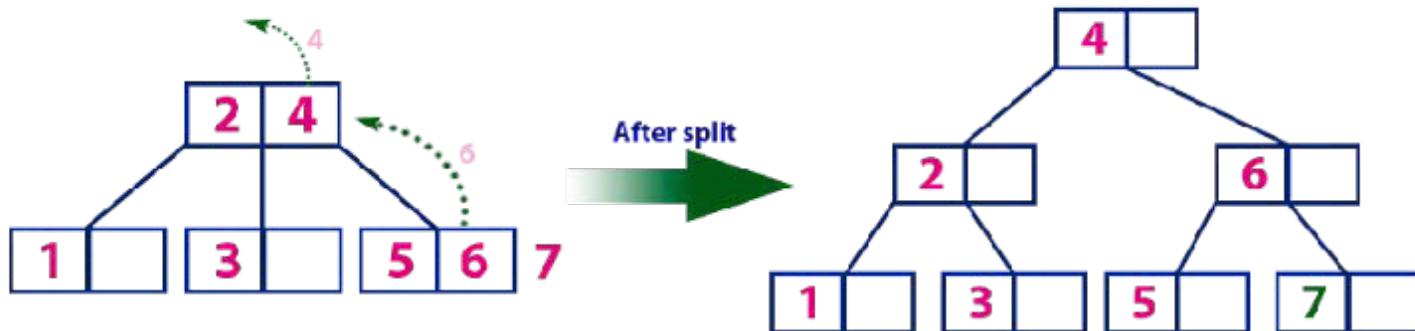

insert(6)

Element '6' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node with value '5' and it has an empty position. So, new element (6) can be inserted at that empty position.



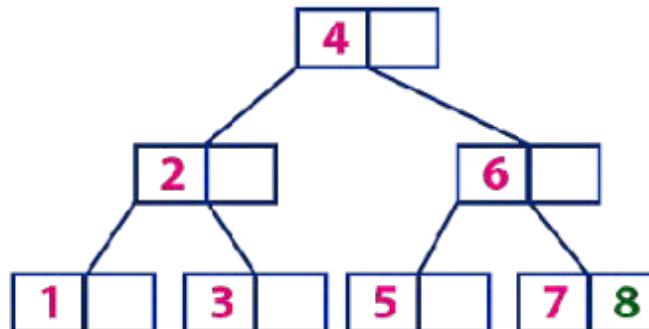
insert(7)

Element '7' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (6) to its parent node (2&4). But the parent (2&4) is also full. So, again we split the node (2&4) by sending middle value '4' to its parent but this node doesn't have parent. So, the element '4' becomes new root node for the tree.



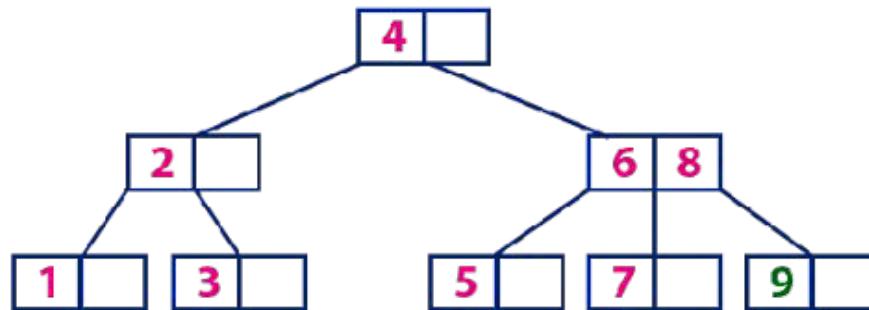
insert(8)

Element '8' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '8' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7) and it has an empty position. So, new element (8) can be inserted at that empty position.

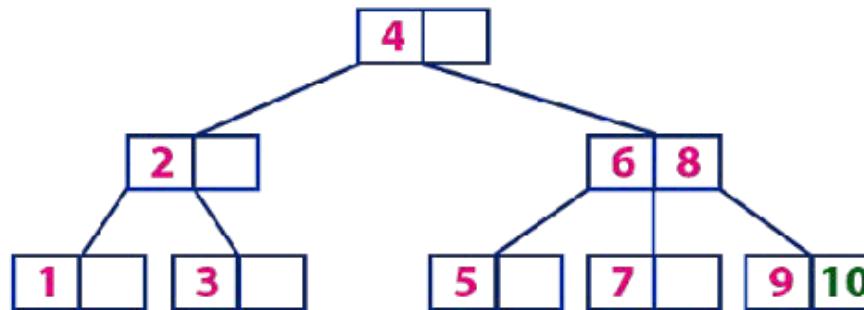


insert(9)

Element '9' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '9' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7 & 8). This leaf node is already full. So, we split this node by sending middle value (8) to its parent node. The parent node (6) has an empty position. So, '8' is added at that position. And new element is added as a new leaf node.


insert(10)

Element '10' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with values '6 & 8'. '10' is larger than '6 & 8' and it is also not a leaf node. So, we move to the right of '8'. We reach to a leaf node (9). This leaf node has an empty position. So, new element '10' is added at that empty position.



Construct a B-tree with order 3 for the following set of values

(2, 3, 5, 7, 11, 17, 19, 23, 29, 31)

Assume that the tree is initially empty and values are inserted in given order.

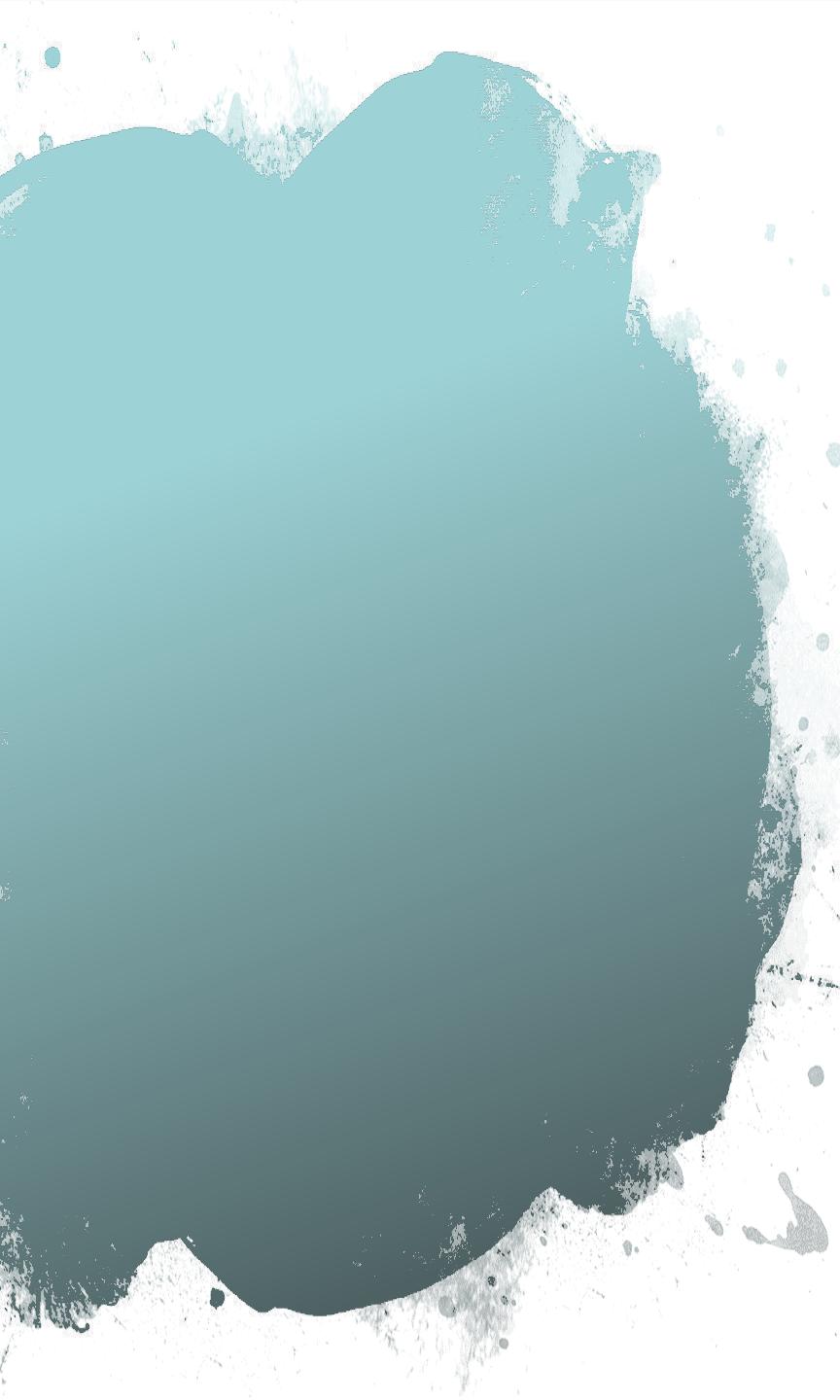
Eg.	Min	max
Keys	1	2
Children	2	3

(2, 3, 5, 7, 11, 17, 19, 23, 29, 31)

Order = 4

Index = 60,70,75,51,52,65,68,77,78,79





DATA STRUCTURES USING C

Module 4 - Lecture VIII

Trees

Prepared By
Ms. Smriti Sehgal

Deleting an element on a B-tree consists of three main events:

- **searching the node where the key to be deleted exists**
- deleting the key
- balancing the tree, if required

While deleting a node from a tree, a condition called **underflow** may occur. Underflow occurs when a node contains less than the minimum number of keys it should hold.

Important Terms:

Inorder Predecessor

The largest key on the left child of a node is called its inorder predecessor.

Inorder Successor

The smallest key on the right child of a node is called its inorder successor.

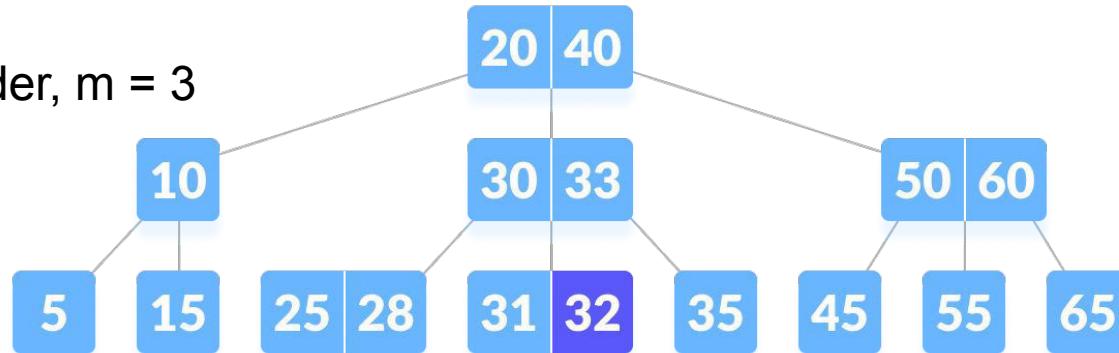
B tree of degree m has **following properties.**

- A node can have a maximum of m children.
- A node can contain a maximum of $m - 1$ keys.
- A node should have a minimum of $\lceil m/2 \rceil$ children.
- A node (except root node) should contain a minimum of $\lceil m/2 \rceil - 1$ key.

Case 1: Element to be deleted is in leaf

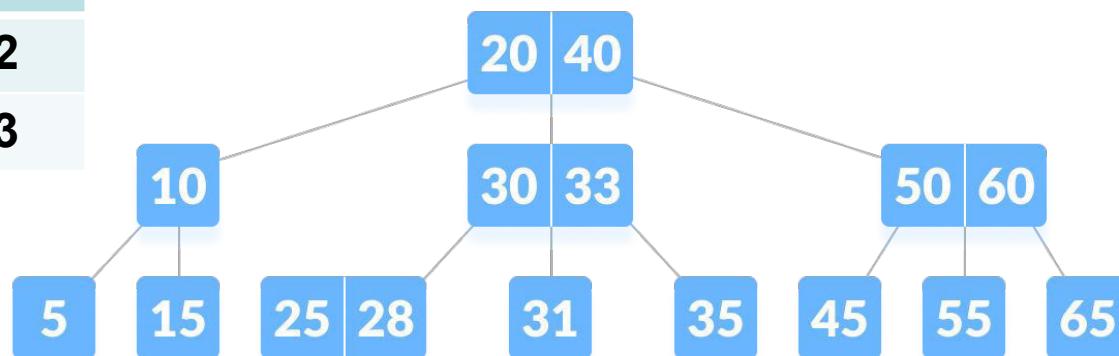
case 1, sub-case 1: The deletion of the key does not violate the property of the minimum number of keys a node should hold.

Consider B-Tree of order, $m = 3$



delete 32

Except Root	Min	Max
Keys	1	2
Children	2	3

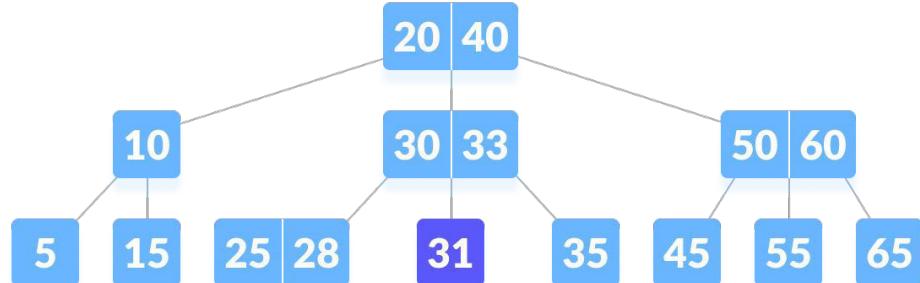


case 1, sub-case 2: The deletion of the key **violates the property** of the minimum number of keys a node should hold.

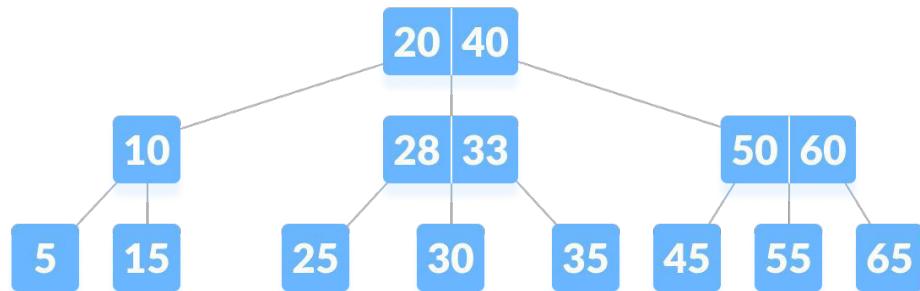
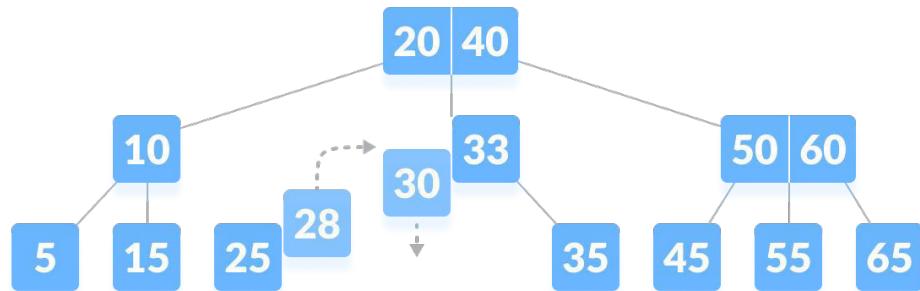
Solution: we borrow a key from its immediate neighbouring sibling node in the order of left to right.

First, visit the immediate left sibling. If the left sibling node has more than a minimum number of keys, then borrow a key from this node.

Else, check to borrow from the immediate right sibling node.

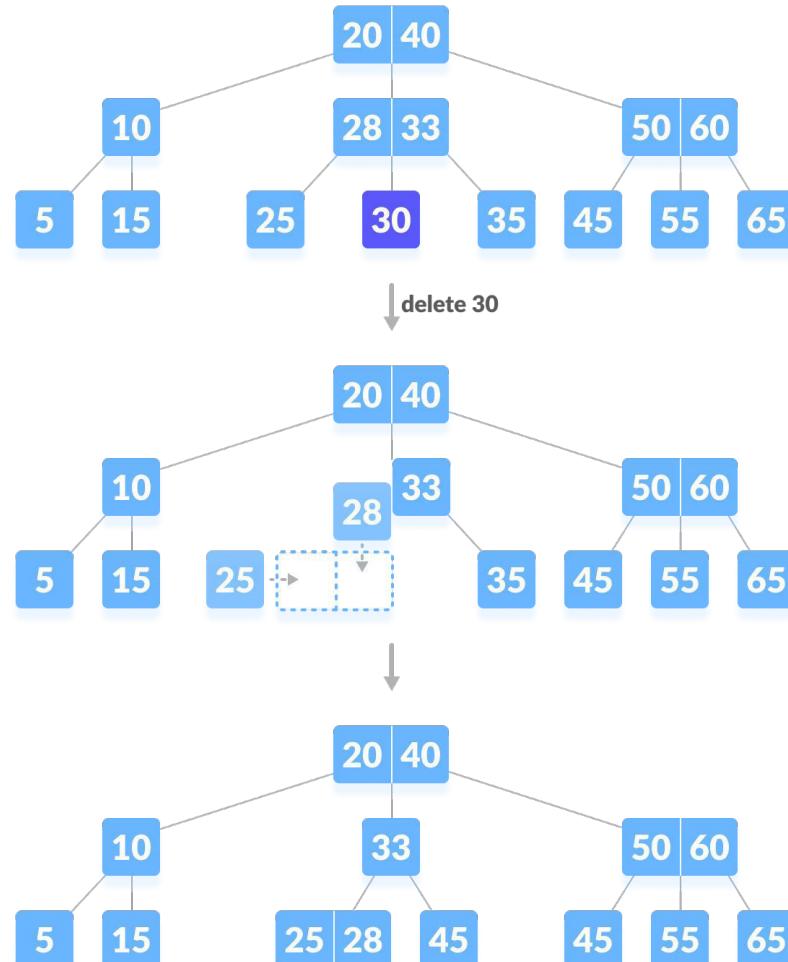


delete 31



If both the immediate sibling nodes already have a minimum number of keys, then merge the node with either the left sibling node or the right sibling node. **This merging is done through the parent node.**

Except Root	Min	Max
Keys	1	2
Children	2	3

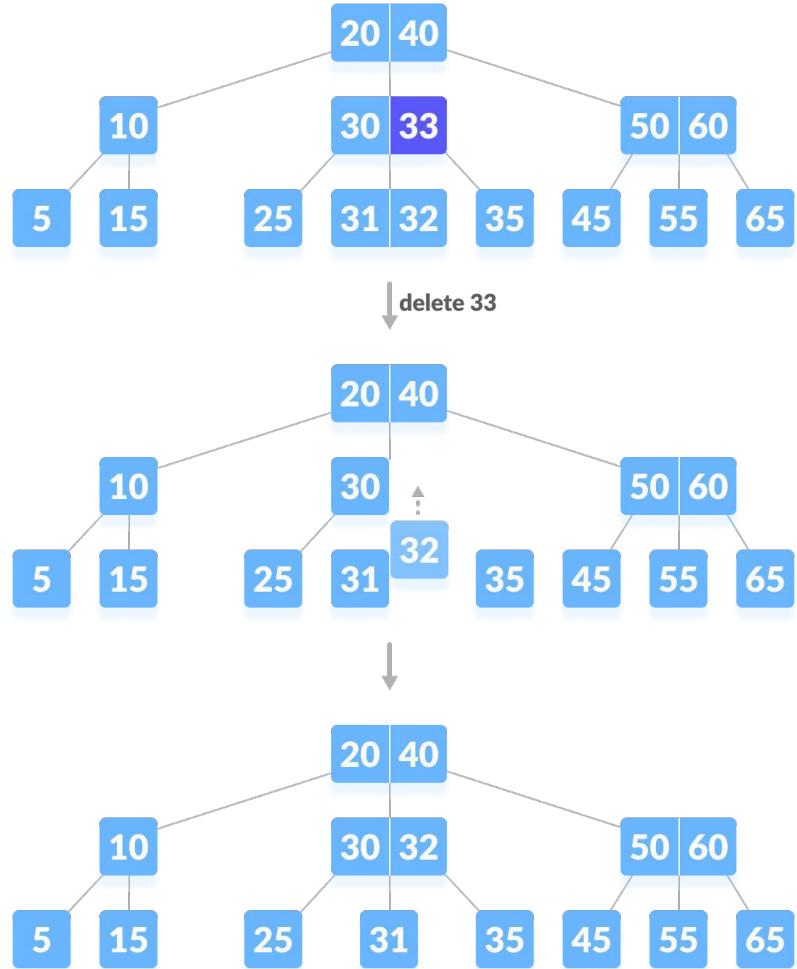


Case 2: Element to be deleted is internal node

Case 2, Sub-case 1:

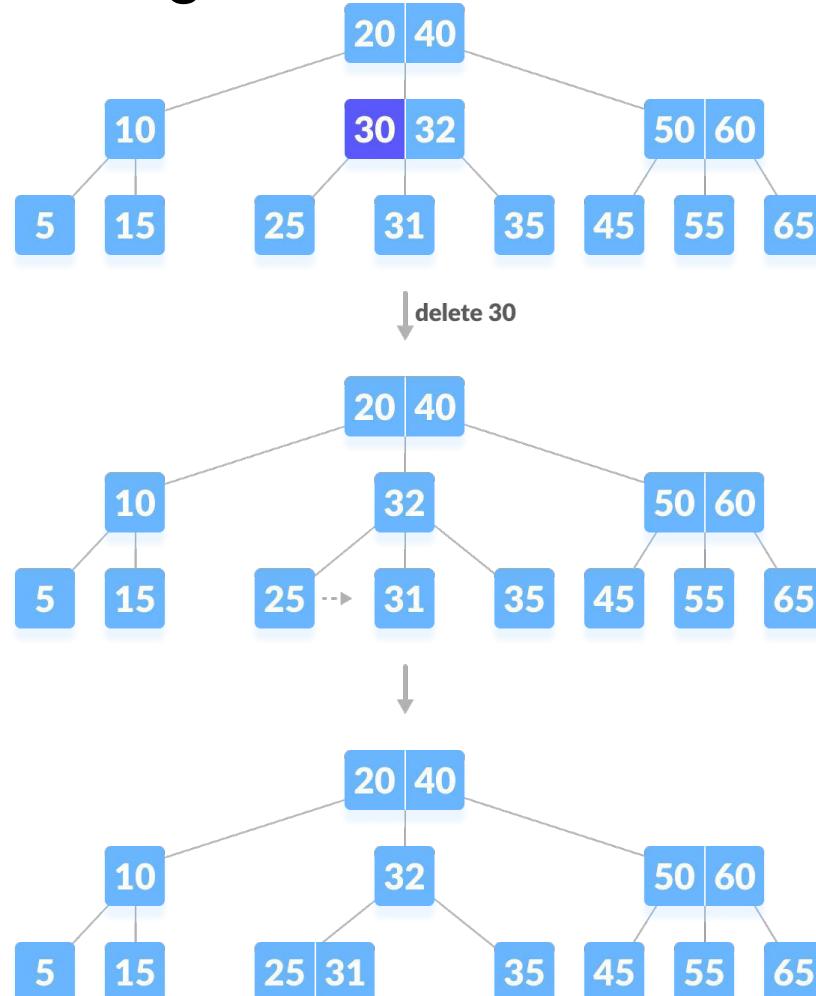
The internal node, which is deleted, is replaced by an inorder predecessor if the left child has more than the minimum number of keys.

Except Root	Min	Max
Keys	1	2
Children	2	3

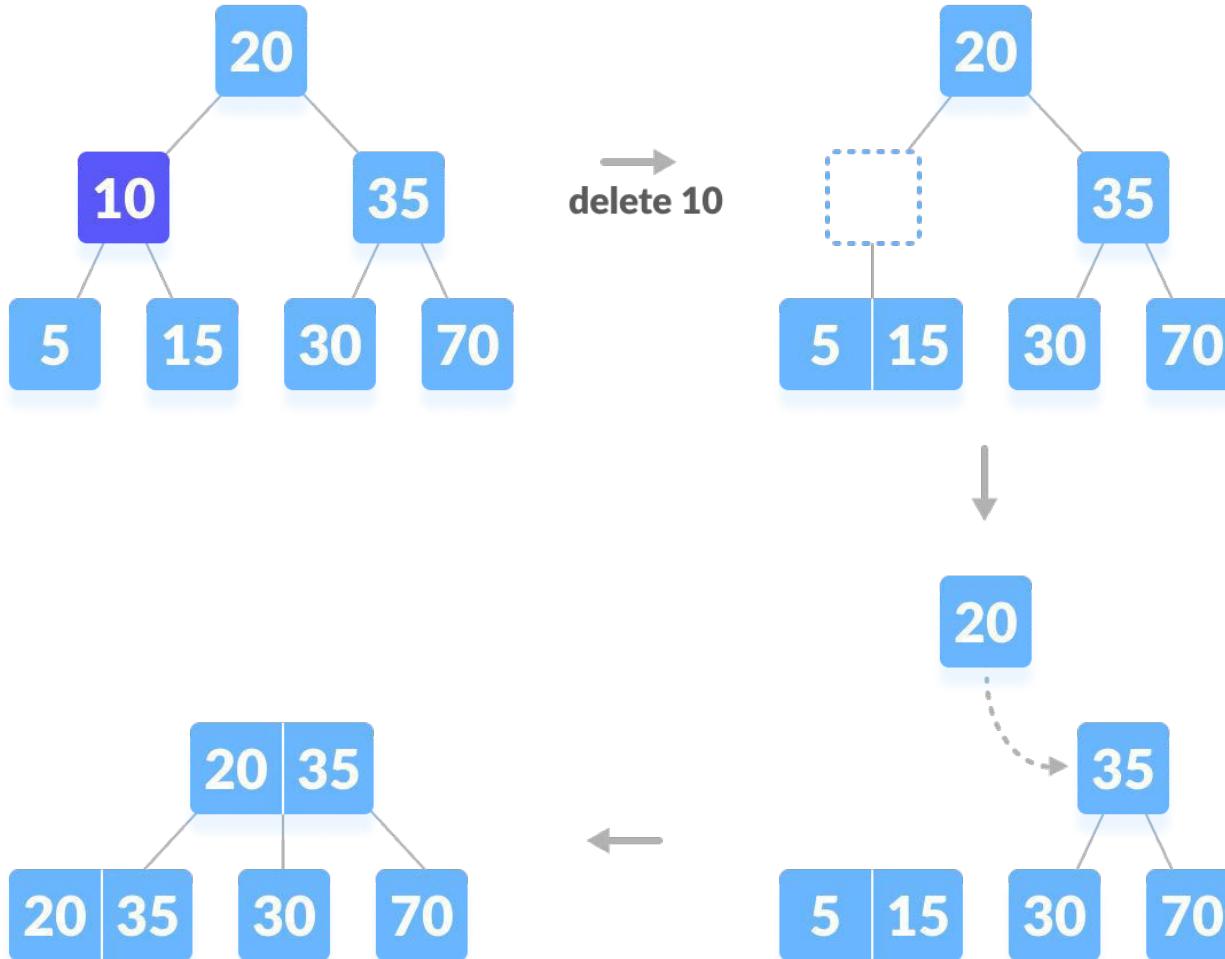


If either child has exactly a minimum number of keys then, merge the left and the right children.

Except Root	Min	Max
Keys	1	2
Children	2	3



Case 3: Element to be deleted is internal node



Create a
B-Tree
using
following
indexes:

Order = 4

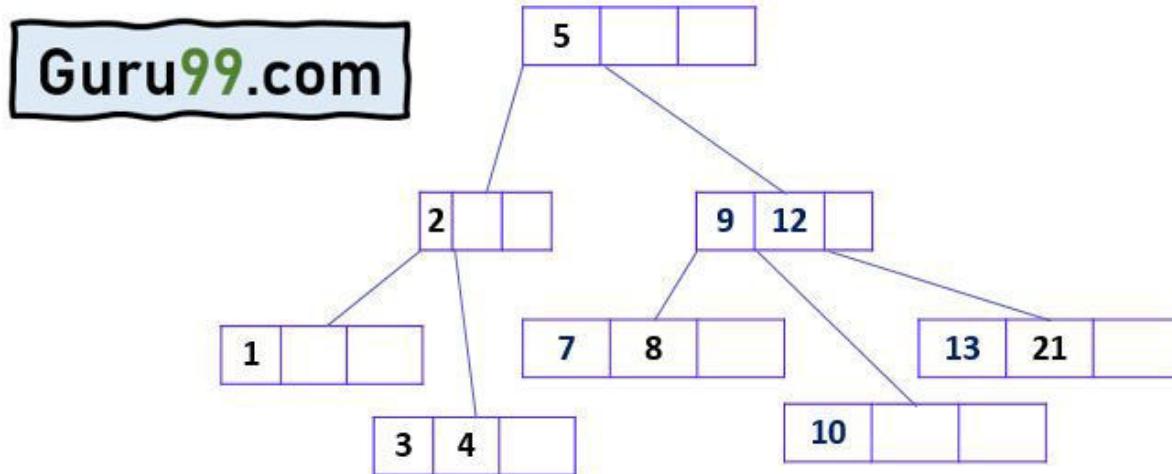
Index = 5,3,21,9,1,13,2,7,10,12,4,8

EXAMPLE SOLVED

Order = 4

Index = 5,3,21,9,1,13,2,7,10,12,4,8

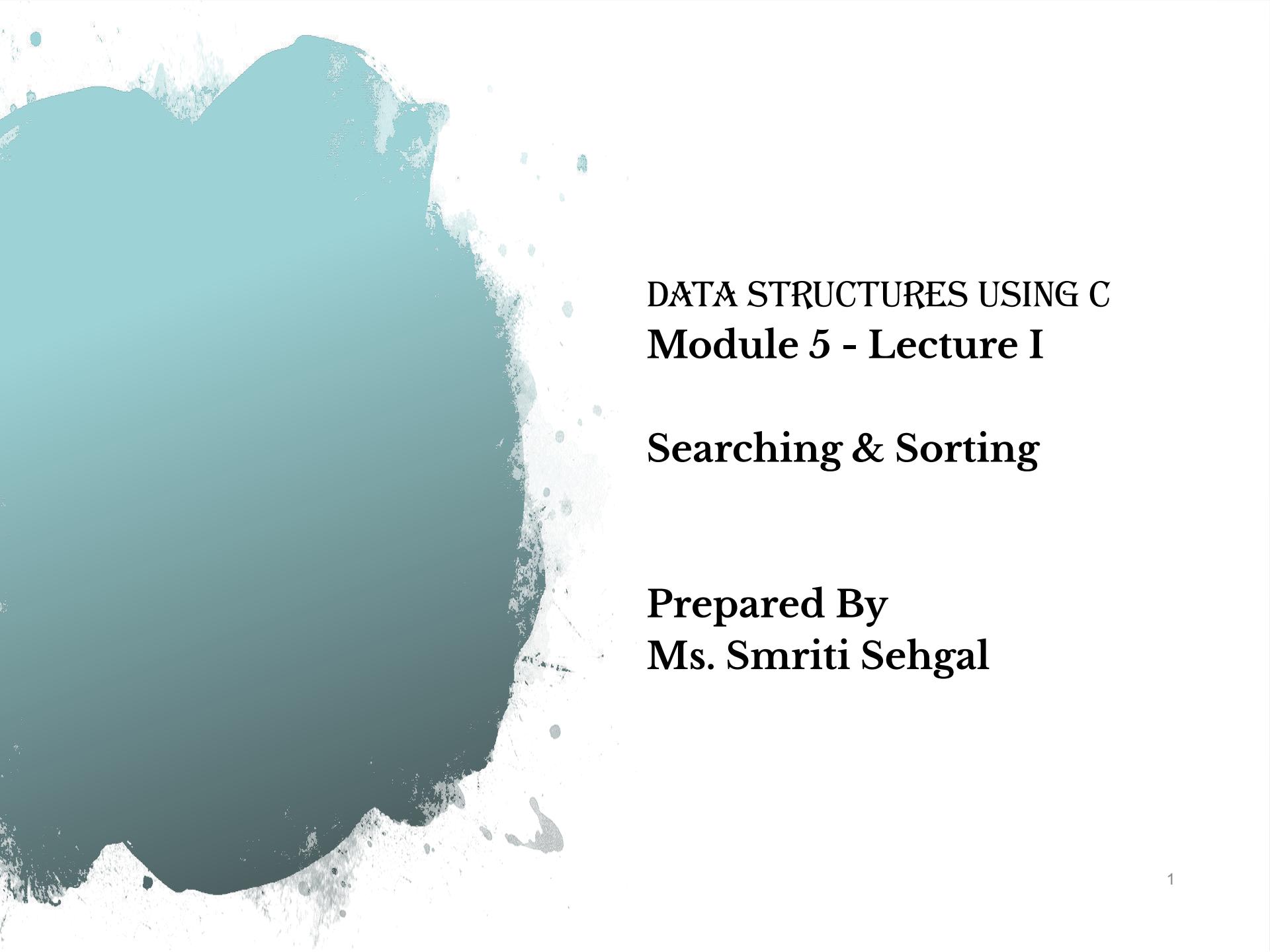
Except Root	Min	Max
Keys	1	3
Children	2	4



Create a B-Tree with following values:

3,14,7,1,8,5,11,17,13,6,23,12,20,26,4,16,1
8,24,25,19





DATA STRUCTURES USING C

Module 5 - Lecture I

Searching & Sorting

Prepared By
Ms. Smriti Sehgal

Syllabus

- **Insertion Sort, Bubble sort, Selection sort, Quick sort, Merge sort, Heap sort, Partition exchange sort, Shell sort, Sorting on different keys, External sorting. Linear search, Binary search, Hashing: Hash Functions, Collision Resolution Techniques.**

In this session

We will talk about

- Insertion sort
- Bubble sort
- Selection sort

Insertion Sort

Amity School of Engineering & Technology

Insertion sort is a simple comparison based sorting algorithm that works similar to the way you sort playing cards in your hands.

The array is virtually split into a sorted and an unsorted part.

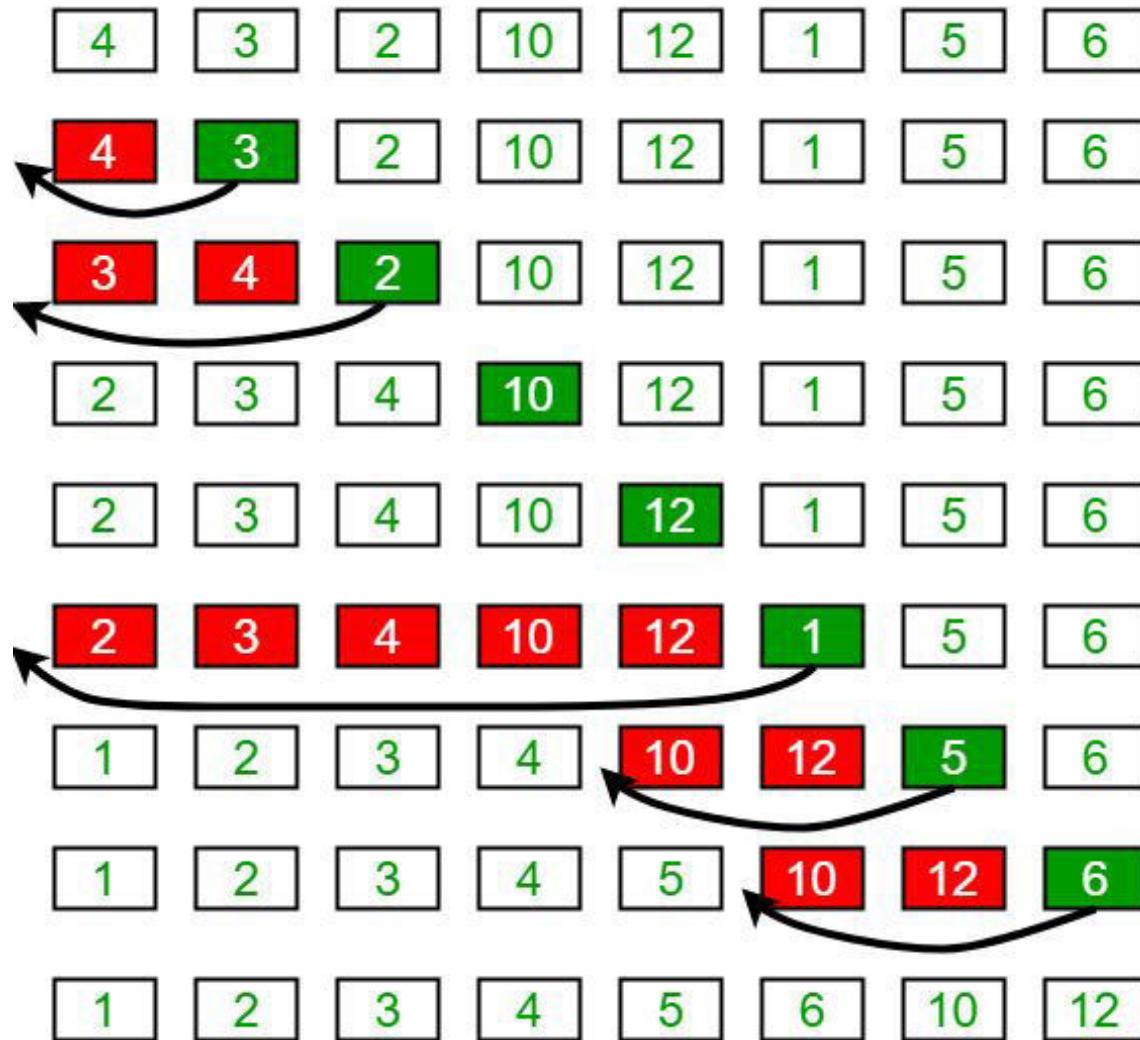
Values from the unsorted part are picked and placed at the correct position in the sorted part.

Algorithm

To sort an array of size n in ascending order:

- 1: Iterate from arr[1] to arr[n] over the array.
- 2: Compare the current element (key) to its predecessor.
- 3: If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

Insertion Sort Execution Example



Sort this using Insertion Sort

8	12	5	4	6	9	3
---	----	---	---	---	---	---

--	--	--	--	--	--	--

--	--	--	--	--	--	--

--	--	--	--	--	--	--

--	--	--	--	--	--	--

Sort this using Insertion Sort

8	12	5	4	6	9	3
---	----	---	---	---	---	---

8	12	5	4	6	9	3
---	----	---	---	---	---	---

8	12	5	4	6	9	3
---	----	---	---	---	---	---

8	5	12	4	6	9	3
---	---	----	---	---	---	---

5	8	12	4	6	9	3
---	---	----	---	---	---	---

Sort this using Insertion Sort

5	8	12	4	6	9	3
---	---	----	---	---	---	---

4	5	8	12	6	9	3
---	---	---	----	---	---	---

4	5	6	8	12	9	3
---	---	---	---	----	---	---

4	5	6	8	9	12	3
---	---	---	---	---	----	---

3	4	5	6	8	9	12
---	---	---	---	---	---	----

Time Complexity: $O(n^2)$

Insertion Sort: Time Complexity

$$\begin{aligned}
T(n) &= \Theta\left(\sum_{i=2}^n (1 + t_i + 1)\right) \\
&= \Theta\left(n + \sum_{i=2}^n t_i\right) \\
&= \Theta\left(n + \sum_{i=2}^n i\right) \\
&= \Theta\left(n + n^2\right) \\
&= \Theta\left(n^2\right).
\end{aligned}$$

Algorithm InsertionSort($A[1..n]$)

```

for  $i \leftarrow 2 \dots n$  do
    L1:  $A[1..i-1]$  is sorted,  $A[i..n]$  is untouched.
    § insert  $A[i]$  into sorted prefix  $A[1..i-1]$  by right-cyclic-shift:
        2.   key  $\leftarrow A[i]$ 
        3.    $j \leftarrow i-1$ 
        4.   while  $j > 0$  and  $A[j] > key$  do
            5.        $A[j+1] \leftarrow A[j]$ 
            6.        $j \leftarrow j-1$ 
        7.   end-while
        8.    $A[j+1] \leftarrow key$ 
        9. end-for
end

```

Worst-case: $t_i = i$ iterations (reverse sorted).

$$\sum_{i=2}^n i = \frac{n(n+1)}{2} - 1 = \Theta(n^2).$$

- Bubble sort is a simple sorting algorithm.
- This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order.
- This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where n is the number of items.
- **How Bubble Sort Works?**
- We take an unsorted array for our example. Bubble sort takes $O(n^2)$ time so we're keeping it short and precise.



- Bubble sort starts with very first two elements, comparing them to check which one is greater.



In this case, value 33 is greater than 14, so it is already in sorted locations.
Next, we compare 33 with 27.



We find that 27 is smaller than 33 and these two values must be swapped.



The new array should look like this –



Next we compare 33 and 35. We find that both are in already sorted positions.



Then we move to the next two values, 35 and 10.



We know then that 10 is smaller 35. Hence they are not sorted.



We swap these values. We find that we have reached the end of the array.
After one iteration, the array should look like this –



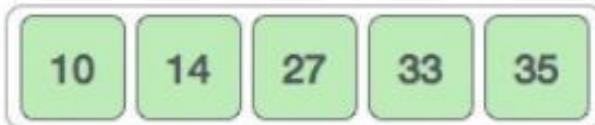
To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this –



Notice that after each iteration, at least one value moves at the end.



And when there's no swap required, bubble sorts learns that an array is completely sorted.



Now we should look into some practical aspects of bubble sort.

Algorithm

We assume **list** is an array of **n** elements. We further assume that **swap** function swaps the values of the given array elements.

```
begin BubbleSort(list)

    for all elements of list
        if list[i] > list[i+1]
            swap(list[i], list[i+1])
        end if
    end for

    return list

end BubbleSort
```

Sort this using Bubble Sort

8	12	5	4	6	9	3
---	----	---	---	---	---	---

--	--	--	--	--	--	--

--	--	--	--	--	--	--

--	--	--	--	--	--	--

--	--	--	--	--	--	--

Sort this using Bubble Sort

8	12	5	4	6	9	3
---	----	---	---	---	---	---

8	5	4	6	9	3	12
---	---	---	---	---	---	----

5	4	6	8	3	9	12
---	---	---	---	---	---	----

4	5	6	3	8	9	12
---	---	---	---	---	---	----

4	5	3	6	8	9	12
---	---	---	---	---	---	----

Sort this using Bubble Sort

4	3	5	6	8	9	12
---	---	---	---	---	---	----

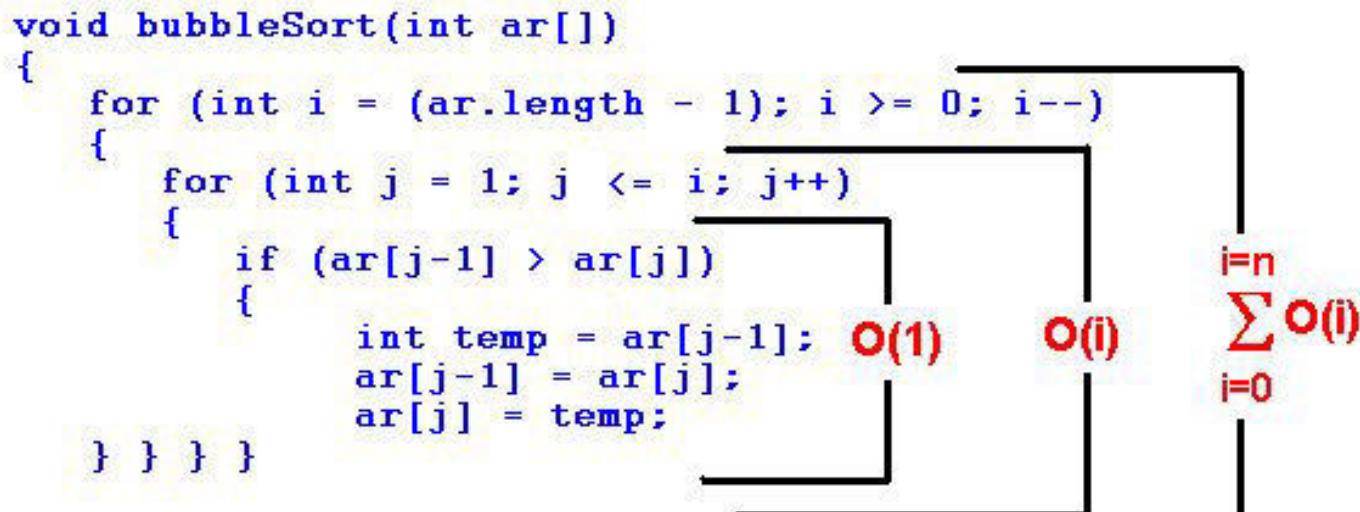
3	4	5	6	8	9	12
---	---	---	---	---	---	----

3	4	5	6	8	9	12
---	---	---	---	---	---	----

```

void bubbleSort(int ar[])
{
    for (int i = (ar.length - 1); i >= 0; i--)
    {
        for (int j = 1; j <= i; j++)
        {
            if (ar[j-1] > ar[j])
            {
                int temp = ar[j-1]; O(1)
                ar[j-1] = ar[j];
                ar[j] = temp;
            }
        }
    }
}

```



$$\sum_{i=0}^{n-1} O(i) = 1 + 2 + 3 + \dots + (n-1) = O(n^2)$$

Selection Sort

Amity School of Engineering & Technology

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning.

The algorithm maintains two subarrays in a given array:

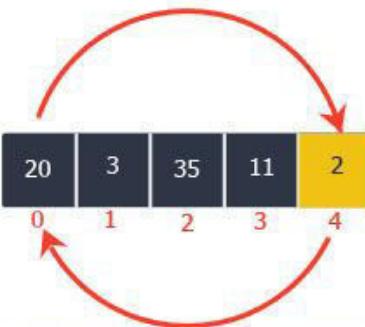
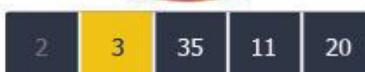
1. The subarray which is already sorted.
2. Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element from the unsorted subarray is picked and moved to the sorted subarray.

Given Array

20	3	35	11	2
0	1	2	3	4

Find minimum element in Array [0 ... 4] and place it at beginning

- a. 
- | | | | | |
|----|---|----|----|---|
| 20 | 3 | 35 | 11 | 2 |
| 0 | 1 | 2 | 3 | 4 |
- Find in [1 ... 4] and Replace
- b. 
- | | | | | |
|---|---|----|----|----|
| 2 | 3 | 35 | 11 | 20 |
| 0 | 1 | 2 | 3 | 4 |
- Find in [1 ... 4] and Replace
- c. 
- | | | | | |
|---|---|----|----|----|
| 2 | 3 | 35 | 11 | 20 |
| 0 | 1 | 2 | 3 | 4 |
- Find in [2 ... 4] and Replace
- d. 
- | | | | | |
|---|---|----|----|----|
| 2 | 3 | 11 | 35 | 20 |
| 0 | 1 | 2 | 3 | 4 |
- Find in [3 ... 4] and Replace
- e. 
- | | | | | |
|---|---|----|----|----|
| 2 | 3 | 11 | 20 | 35 |
| 0 | 1 | 2 | 3 | 4 |
- Sorted Array </>

Analysis of Selection Sort

<i>Alg.:</i> SELECTION-SORT(A)	cost	times
$n \leftarrow \text{length}[A]$	c_1	1
for $j \leftarrow 1$ to $n - 1$	c_2	n
do $\text{smallest} \leftarrow j$	c_3	$n-1$
$\approx n^2/2$ <i>comparisons</i>	c_4	$\sum_{j=1}^{n-1} (n-j+1)$
for $i \leftarrow j + 1$ to n	c_5	$\sum_{j=1}^{n-1} (n-j)$
do if $A[i] < A[\text{smallest}]$	c_6	$\sum_{j=1}^{n-1} (n-j)$
then $\text{smallest} \leftarrow i$	c_7	$n-1$
exchange $A[j] \leftrightarrow A[\text{smallest}]$		

$$T(n) = c_1 + c_2 n + c_3(n-1) + c_4 \sum_{j=1}^{n-1} (n-j+1) + c_5 \sum_{j=1}^{n-1} (n-j) + c_6 \sum_{j=2}^{n-1} (n-j) + c_7(n-1) = \Theta(n^2) \quad 31$$

Work for you....

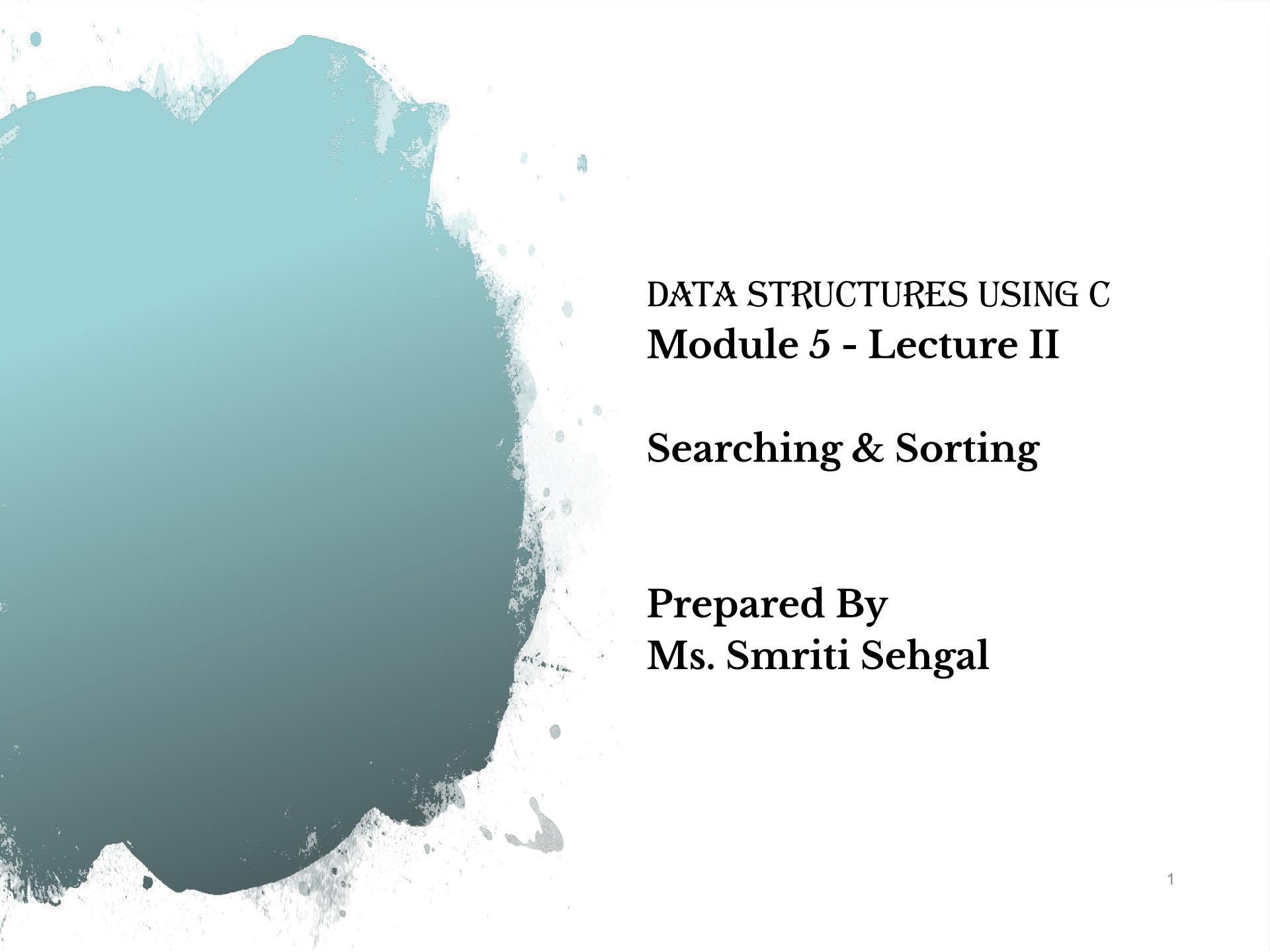
Amity School of Engineering & Technology

- You have been given a pack of cards (Only Spades) numbered 1 to 13 (Ace, 2 to 10, Jack Queen, King) in random order. Write a program to sort these cards. Use **Bubble sort** and print each line of output after every iteration.
- For the same above problem, Write the program using Insertion sort and print each line of output after every iteration.
- Compare the number of steps for each sorting technique

Web Resources

- <https://www.cs.usfca.edu/~galles/visualization/Search.html> (Final) for linear search and binary search
- <https://blog.penjee.com/binary-vs-linear-search-animated-gifs/> (for linear search and binary search comparisons)
- <https://www.cs.usfca.edu/~galles/visualization/Algorithms.html>
- <http://www.cs.armstrong.edu/liang/animation/web/BubbleSort.html> (Bubble Sort)
- http://www.ee.ryerson.ca/~courses/coe428/sorting/bubble_sort.html (bUBBLE SORT)
- <https://courses.cs.vt.edu/csonline/Algorithms/Lessons/InsertionCardSort/insertioncardsort.swf> (Insertion sort final)
- <http://www.ee.ryerson.ca/~courses/coe428/sorting/insertionsort.html> (INSERTION SORT ADDITIONAL)





DATA STRUCTURES USING C

Module 5 - Lecture II

Searching & Sorting

Prepared By
Ms. Smriti Sehgal

Syllabus

- Insertion Sort, Bubble sort, Selection sort, **Quick sort, Merge sort, Heap sort**, Partition exchange sort, Shell sort, Sorting on different keys, External sorting. Linear search, Binary search, Hashing: Hash Functions, Collision Resolution Techniques.

In this session

We will talk about

- Quick sort
- Merge sort
- Heap sort

- Efficient sorting algorithm
- Example of **Divide and Conquer** algorithm
- Two phases
 - Partition phase
 - **Divides** the work into half
 - Sort phase
 - **Conquers** the halves!

- Partition
 - Choose a **pivot**
 - Find the position for the pivot so that
 - all elements to the left are less / equal
 - all elements to the right are greater / equal

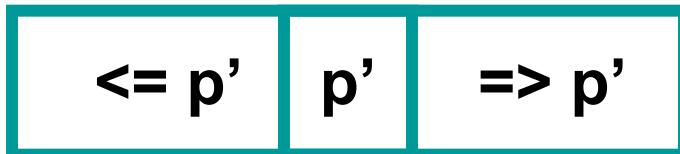
< =pivot

pivot

=> pivot

- Conquer
 - Apply the same algorithm to each half

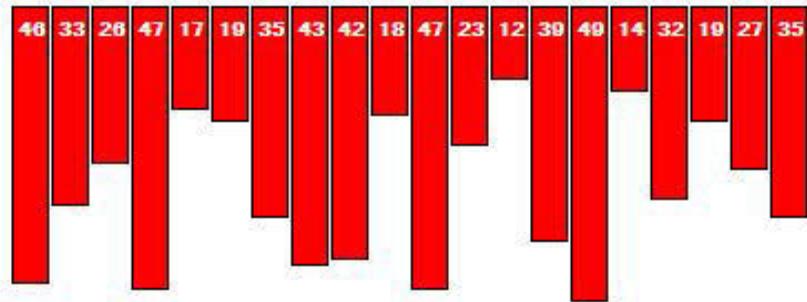
< pivot



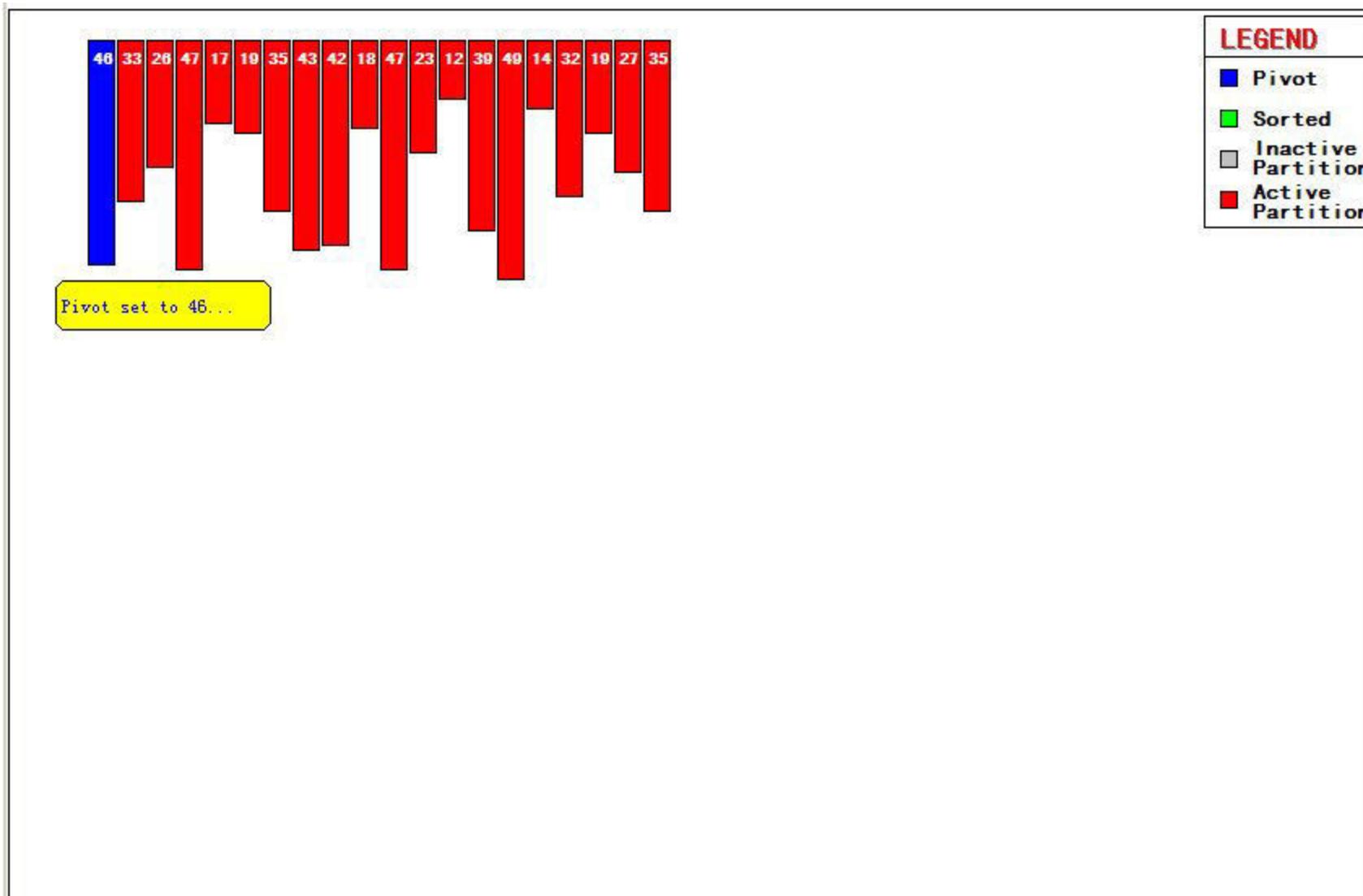
> pivot



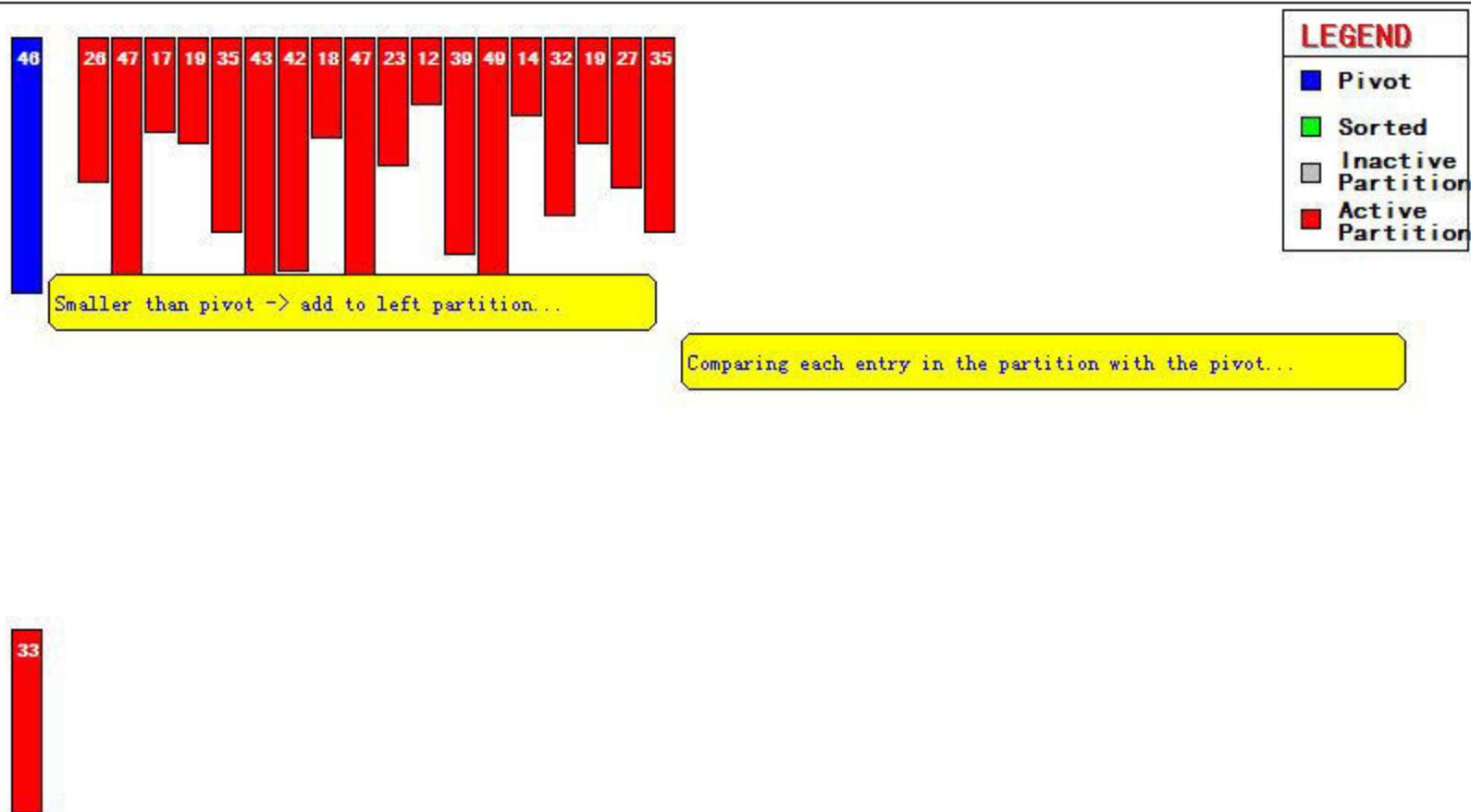
46, 33, 26, 47, 17, 19, 35, 43, 42, 18, 47, 23, 12, 39, 49, 14, 32, 19, 27, 35



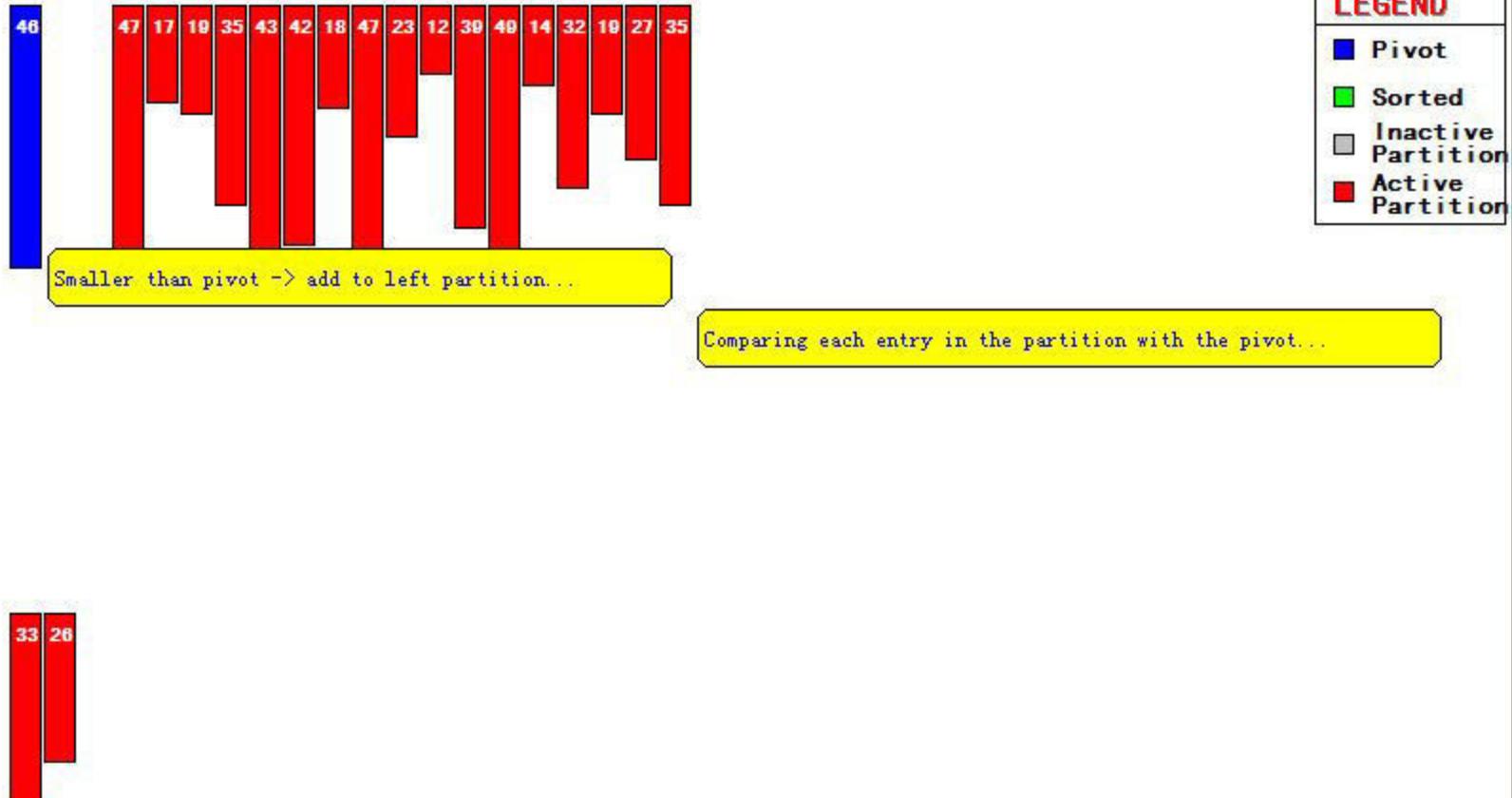
46, 33, 26, 47, 17, 19, 35, 43, 42, 18, 47, 23, 12, 39, 49, 14, 32, 19, 27, 35

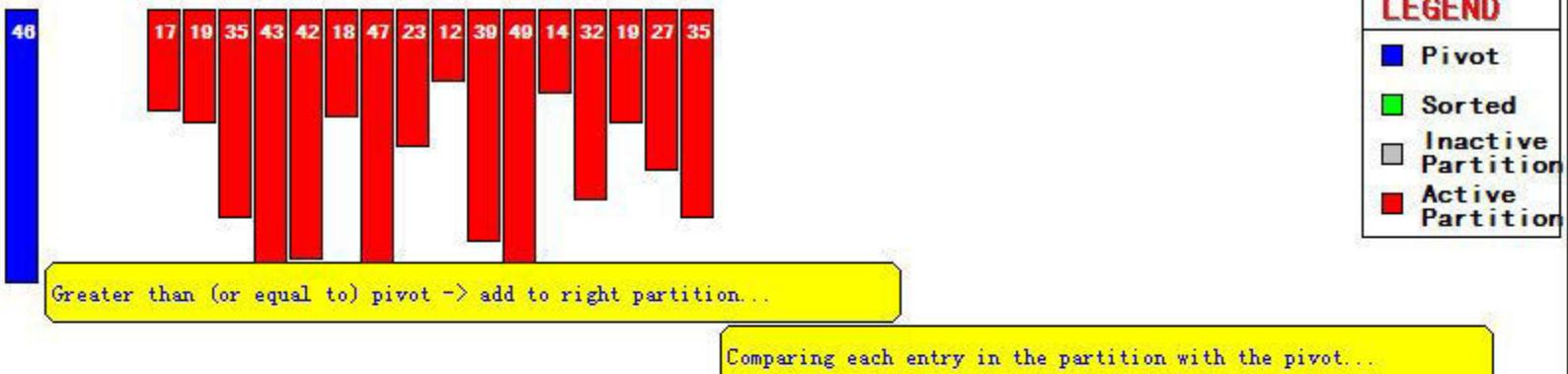


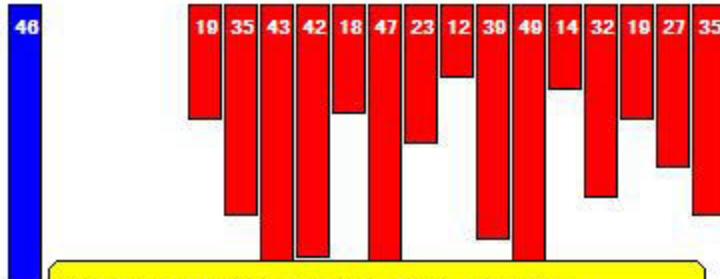
46, 33, 26, 47, 17, 19, 35, 43, 42, 18, 47, 23, 12, 39, 49, 14, 32, 19, 27, 35



46, 33, 26, 47, 17, 19, 35, 43, 42, 18, 47, 23, 12, 39, 49, 14, 32, 19, 27, 35



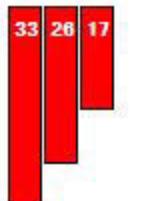




Smaller than pivot -> add to left partition...

LEGEND	
Pivot	■
Sorted	■
Inactive Partition	■
Active Partition	■

Comparing each entry in the partition with the pivot...

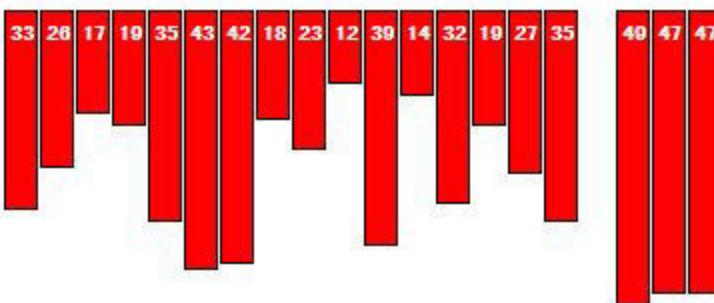


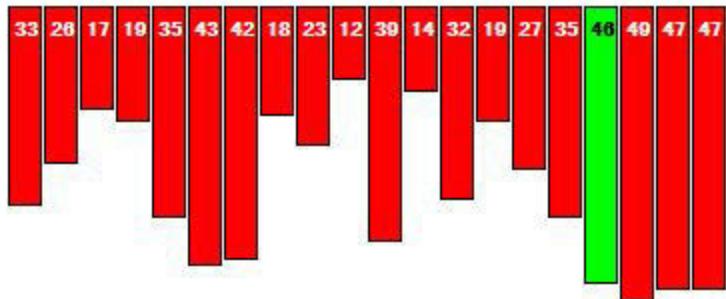
46

Smaller than pivot -> add to left partition...

LEGEND	
Pivot	■
Sorted	■
Inactive Partition	■
Active Partition	■

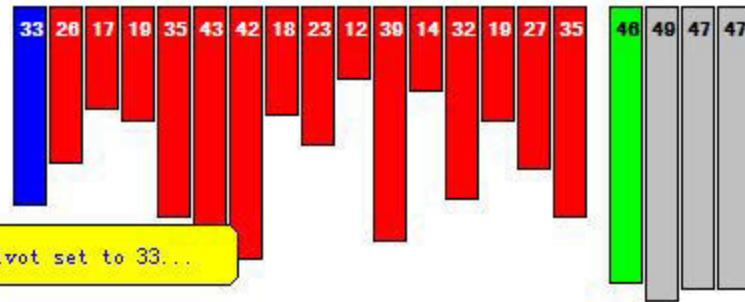
Comparing each entry in the partition with the pivot...



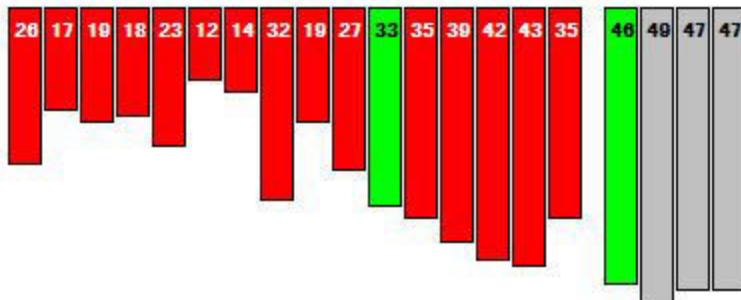


LEGEND

█ Pivot
█ Sorted
█ Inactive Partition
█ Active Partition

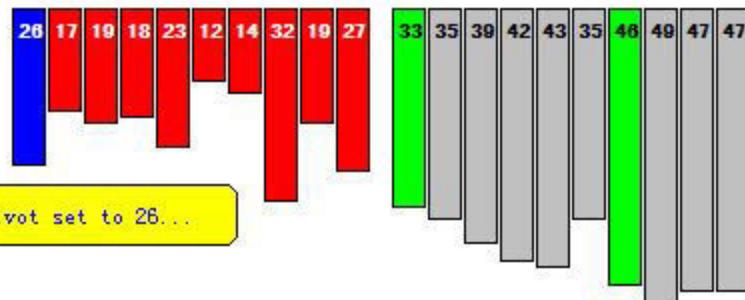


LEGEND	
Pivot	■
Sorted	■
Inactive Partition	■
Active Partition	■

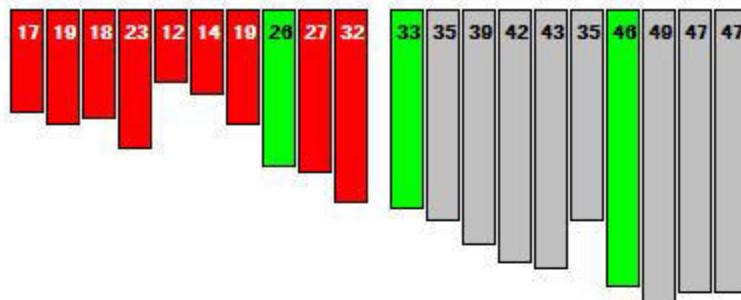


LEGEND

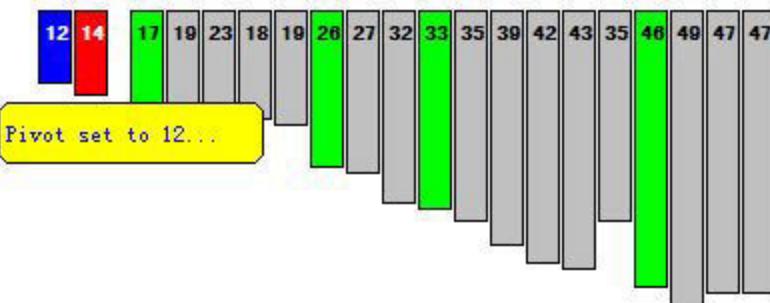
■ Pivot
■ Sorted
■ Inactive Partition
■ Active Partition

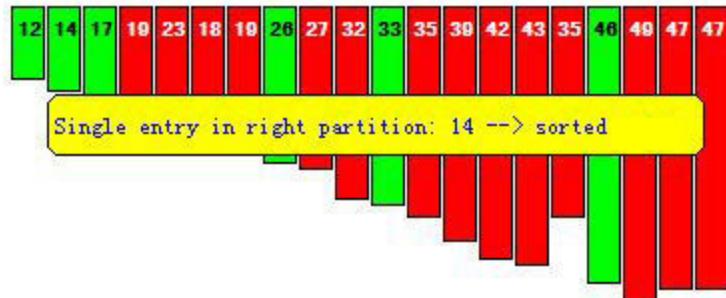


LEGEND	
Pivot	■
Sorted	■
Inactive Partition	■
Active Partition	■

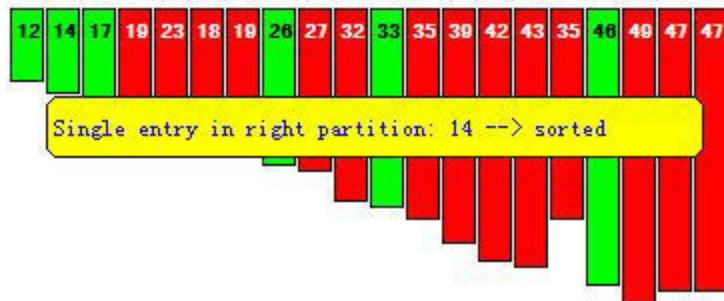


LEGEND	
Pivot	■
Sorted	■
Inactive Partition	■
Active Partition	■

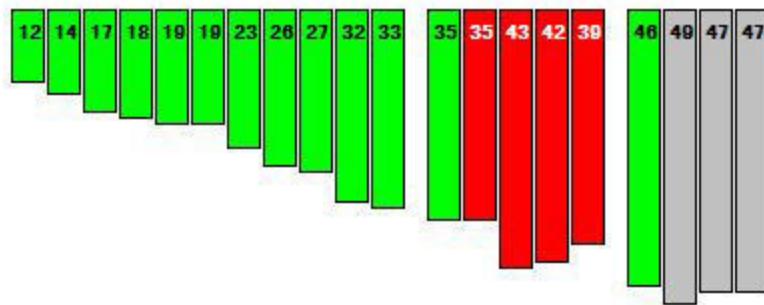




LEGEND	
Pivot	
Sorted	
Inactive Partition	
Active Partition	

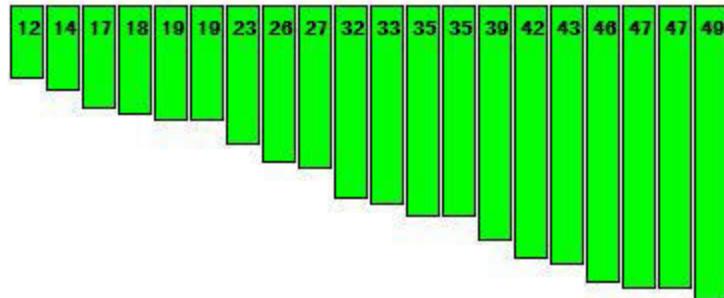


LEGEND	
Pivot	
Sorted	
Inactive Partition	
Partition	
Active Partition	



LEGEND

- Pivot
- Sorted
- Inactive Partition
- Active Partition



- Implementation

```
quicksort( void *a, int low, int high )
{
    int pivot;
    /* Termination condition! */
    if ( high > low )
    {
        pivot = partition( a, low, high );
        quicksort( a, low, pivot-1 );
        quicksort( a, pivot+1, high );
    }
}
```

The code is annotated with three colored boxes: a yellow box labeled "Divide" containing the line `pivot = partition(a, low, high);`, a pink box labeled "Conquer" containing the two recursive calls `quicksort(a, low, pivot-1);` and `quicksort(a, pivot+1, high);`, and a white box containing the rest of the code.

Quicksort - Partition

```
int partition( int *a, int low, int high ) {  
    int left, right;  
    int pivot_item;  
    pivot_item = a[low];  
    pivot = left = low;  
    right = high;  
    while ( left < right ) {  
        /* Move left while item < pivot */  
        while( a[left] <= pivot_item ) left++;  
        /* Move right while item > pivot */  
        while( a[right] > pivot_item ) right--;  
        if ( left < right ) {  
            SWAP(a,left,right); left++; right--;  
        }  
        /* right is final position for the pivot */  
        a[low] = a[right];  
        a[right] = pivot_item;  
        return right;  
    }  
}
```

- Partition
 - Check every item once $O(n)$
- Conquer
 - Divide data in half $O(\log_2 n)$
- Total
 - Product $O(n \log n)$
- *Same as Heapsort*
 - quicksort is *generally* faster
 - Fewer comparisons

● Quicksort

- Generally faster
- Sometimes $O(n^2)$
 - Better pivot selection reduces probability
- Use when you want average good performance
 - Commercial applications, Information systems

● Heap Sort

- Generally slower
- **Guaranteed** $O(n \log n)$

- **Definition:** A *sort* algorithm that splits the items to be sorted into *two* groups, *recursively* sorts each group, and *merge* them into a final sorted sequence. Run time is $O(n \log n)$.

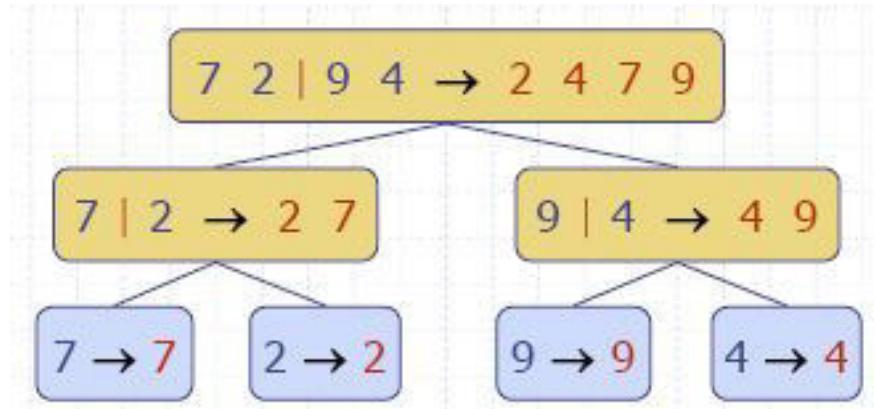
Merge Sort – Divide-and-Conquer

- **Divide-and-conquer** is a general algorithm design paradigm:
 - **Divide**: divide the input data S in two disjoint subsets S_1 and S_2
 - **Conquer**: solve the sub-problems associated with S_1 and S_2 recursively
 - **Combine**: combine the solutions for S_1 and S_2 into a solution for S
- The base case for the recursion are sub-problems of size 0 or 1

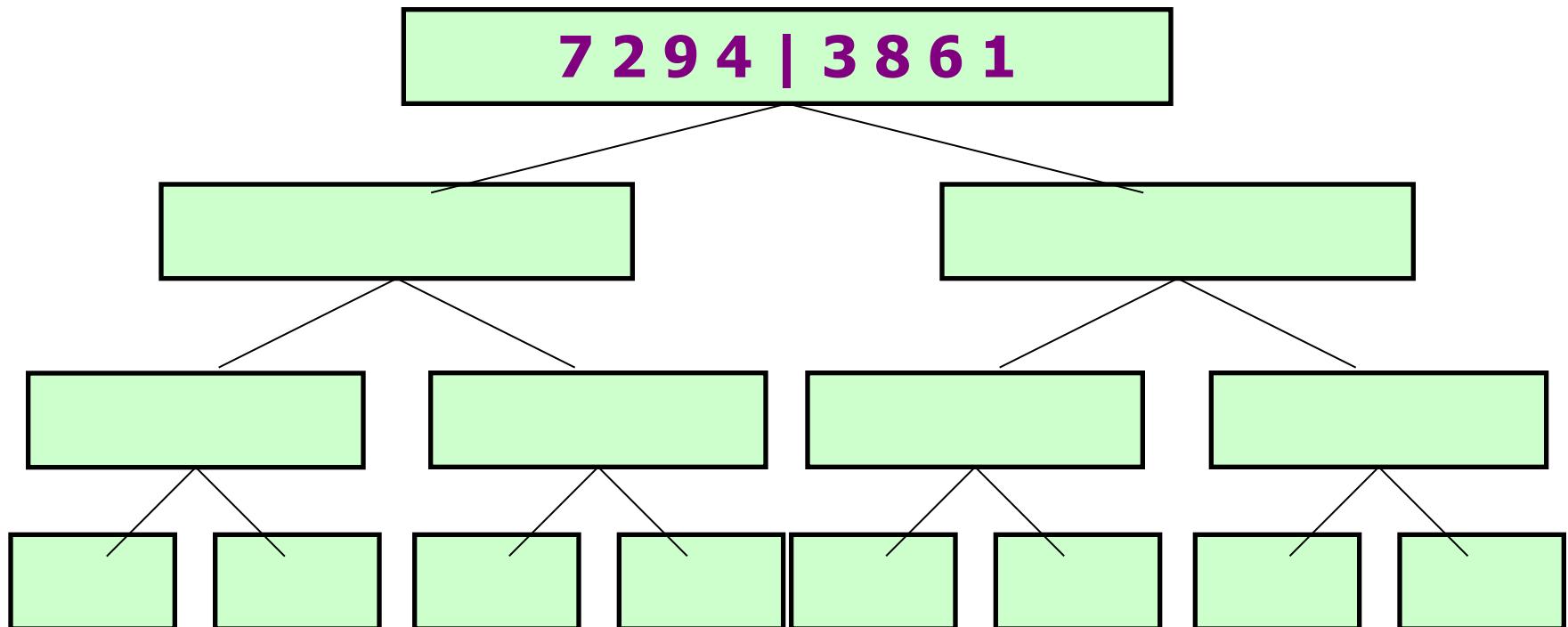
Merge Sort Tree

Amity School of Engineering & Technology

- An execution of merge-sort is depicted by a binary tree
 - each node represents a recursive call of merge-sort and stores
 - unsorted sequence before the execution
 - sorted sequence at the end of the execution
 - the root is the **initial** call
 - the leaves are calls on subsequences of size 0 or 1



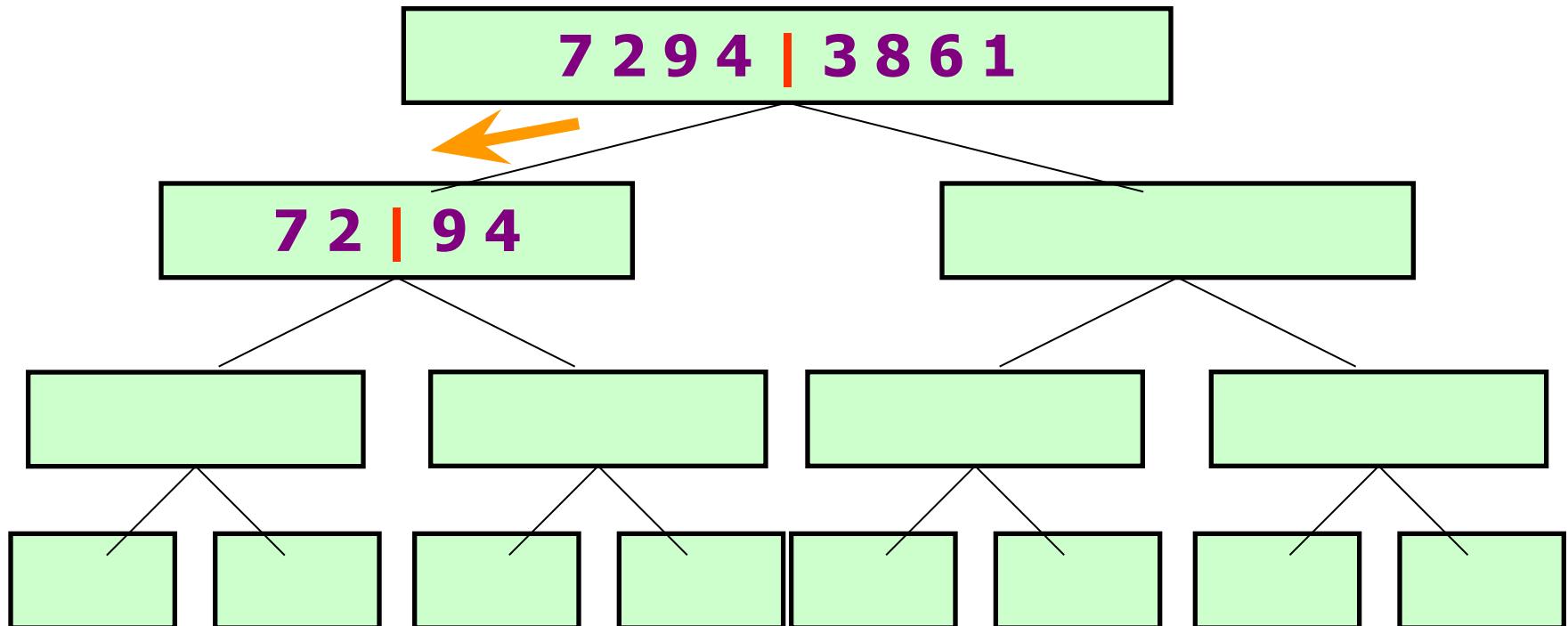
Partition



Merge Sort - example

Amity School of Engineering & Technology

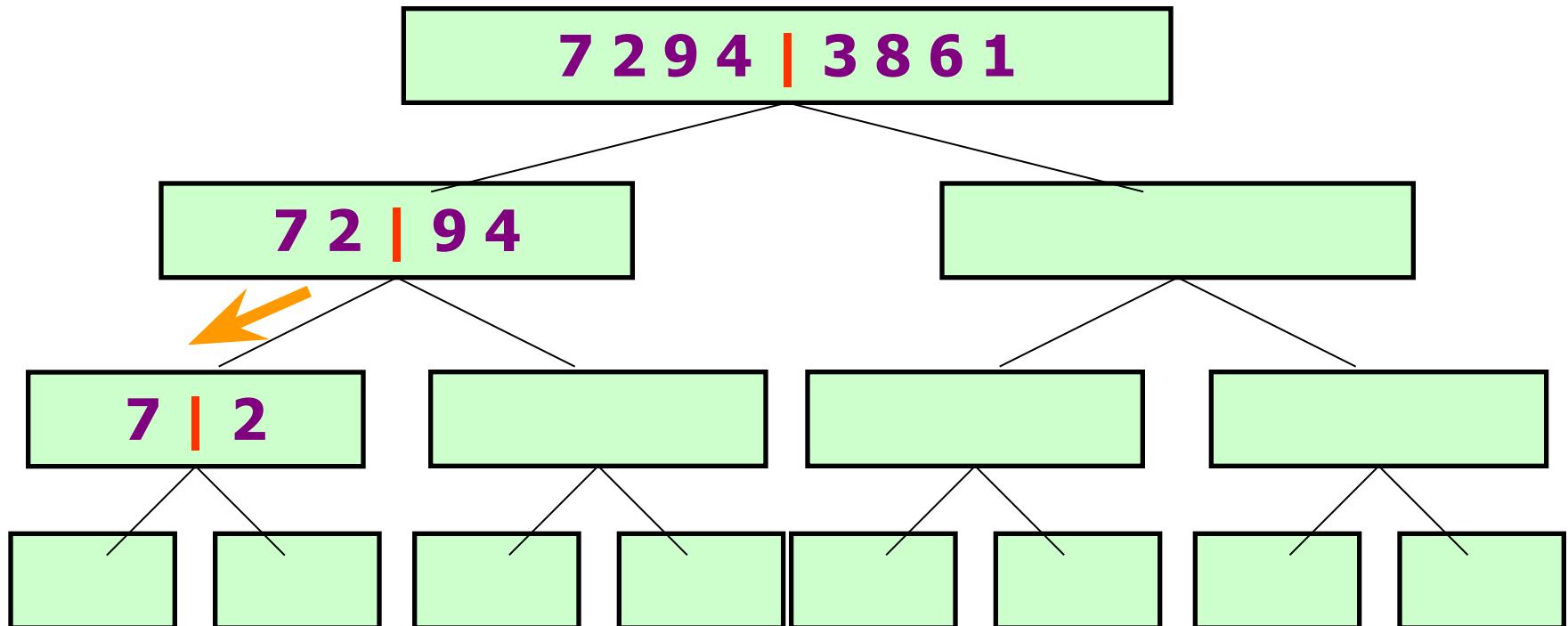
Recursive call, partition



Merge Sort - example

Amity School of Engineering & Technology

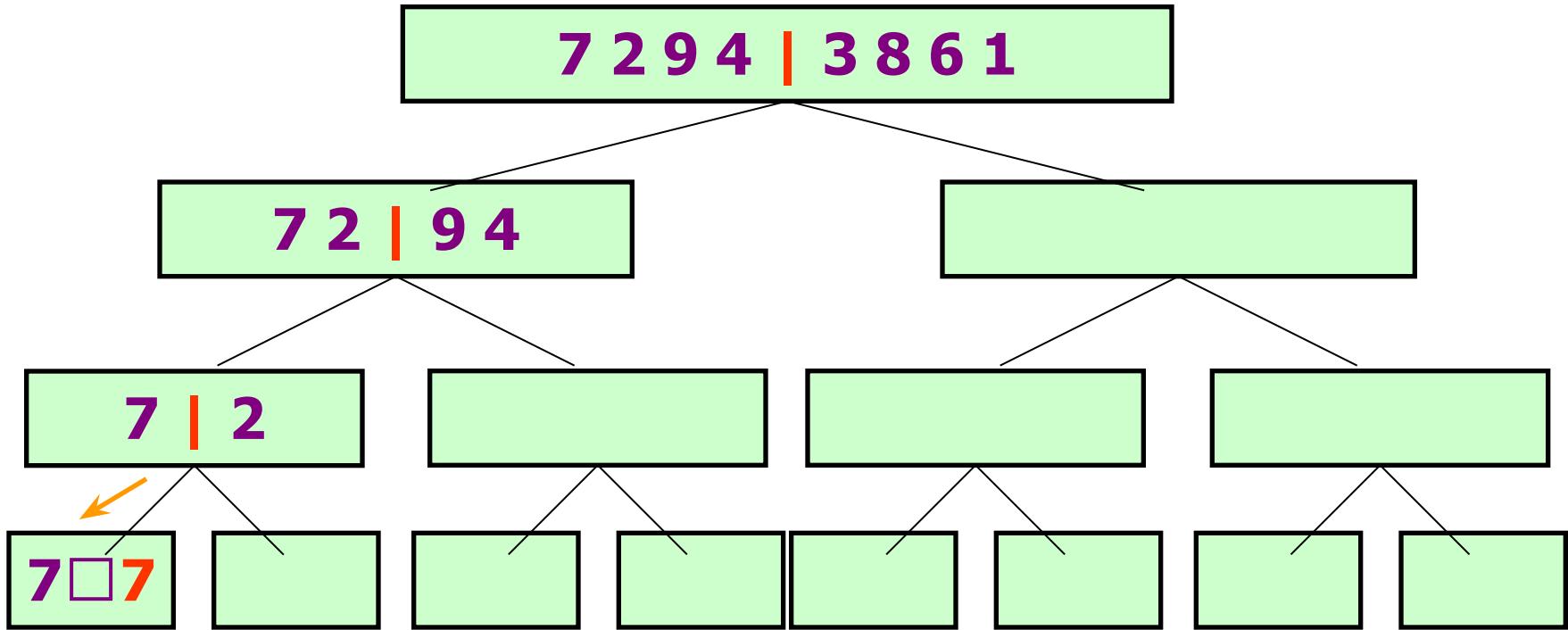
Recursive call, partition



Merge Sort - example

Amity School of Engineering & Technology

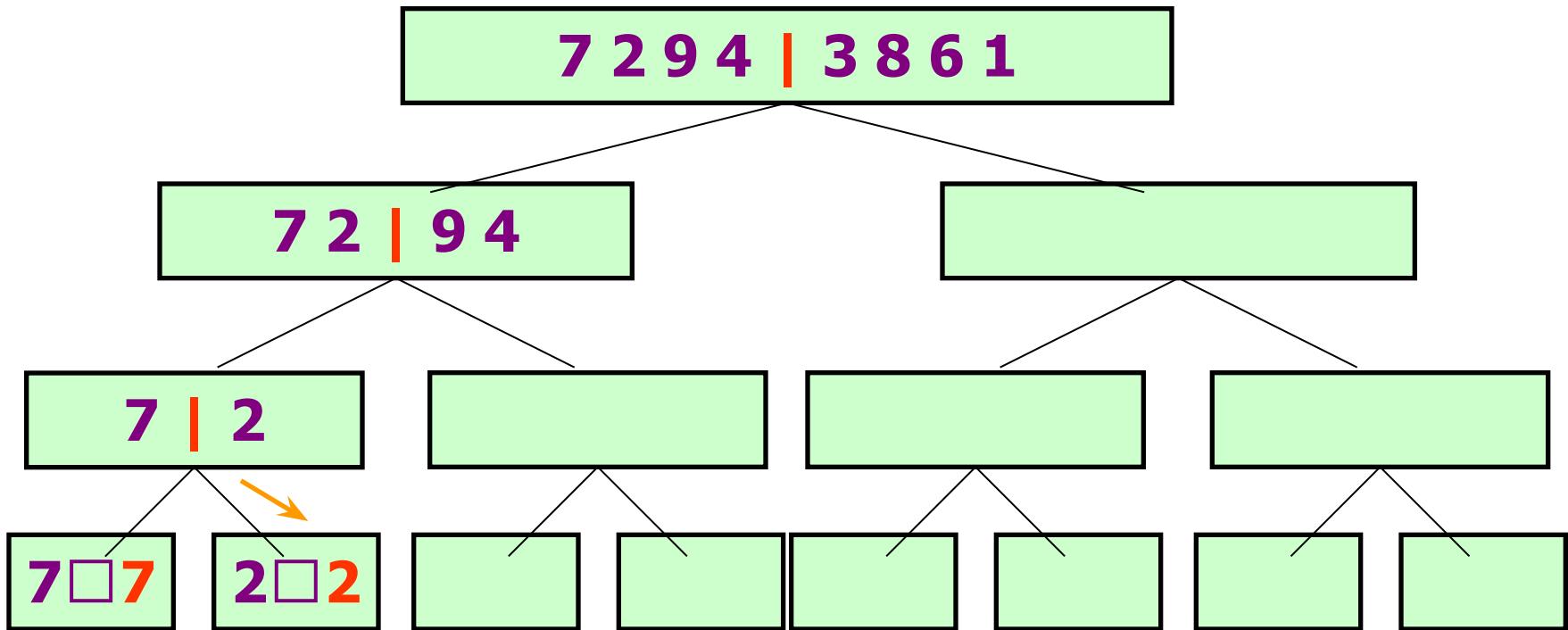
Recursive call, base case



Merge Sort - example

Amity School of Engineering & Technology

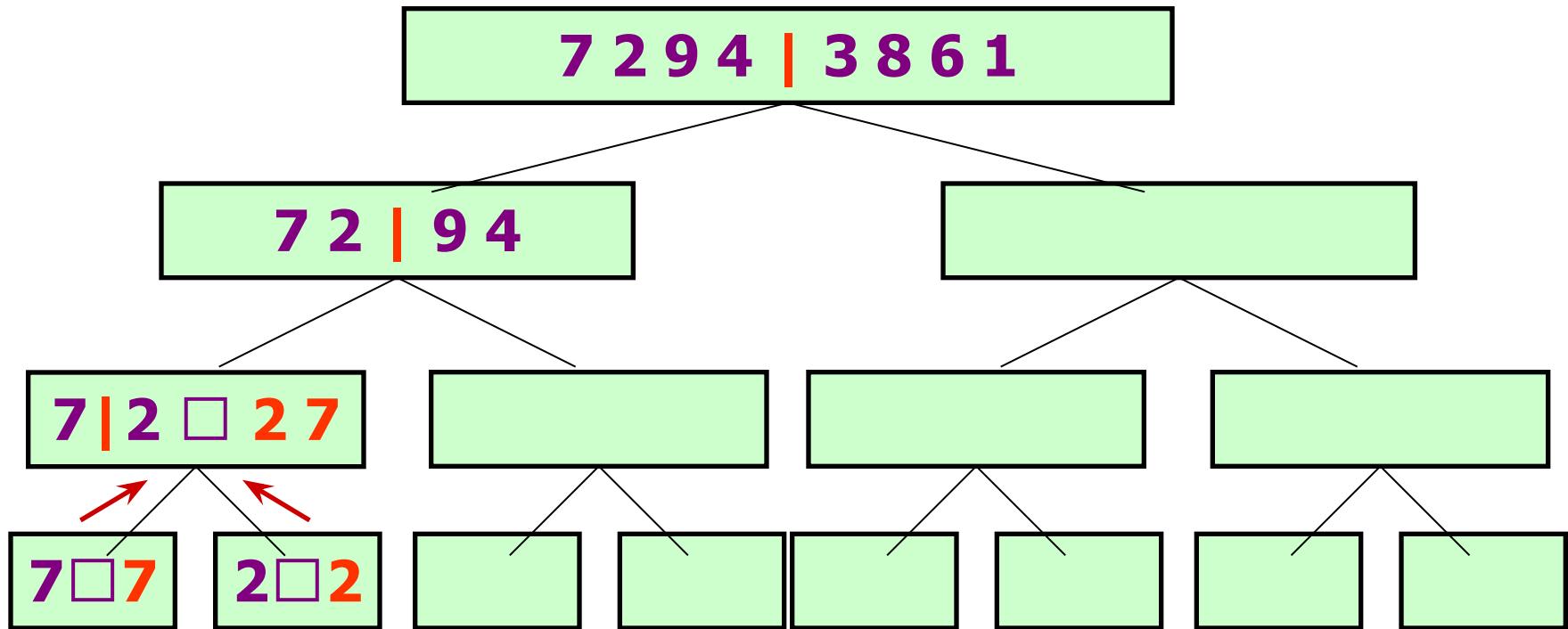
Recursive call, base case



Merge Sort - example

Amity School of Engineering & Technology

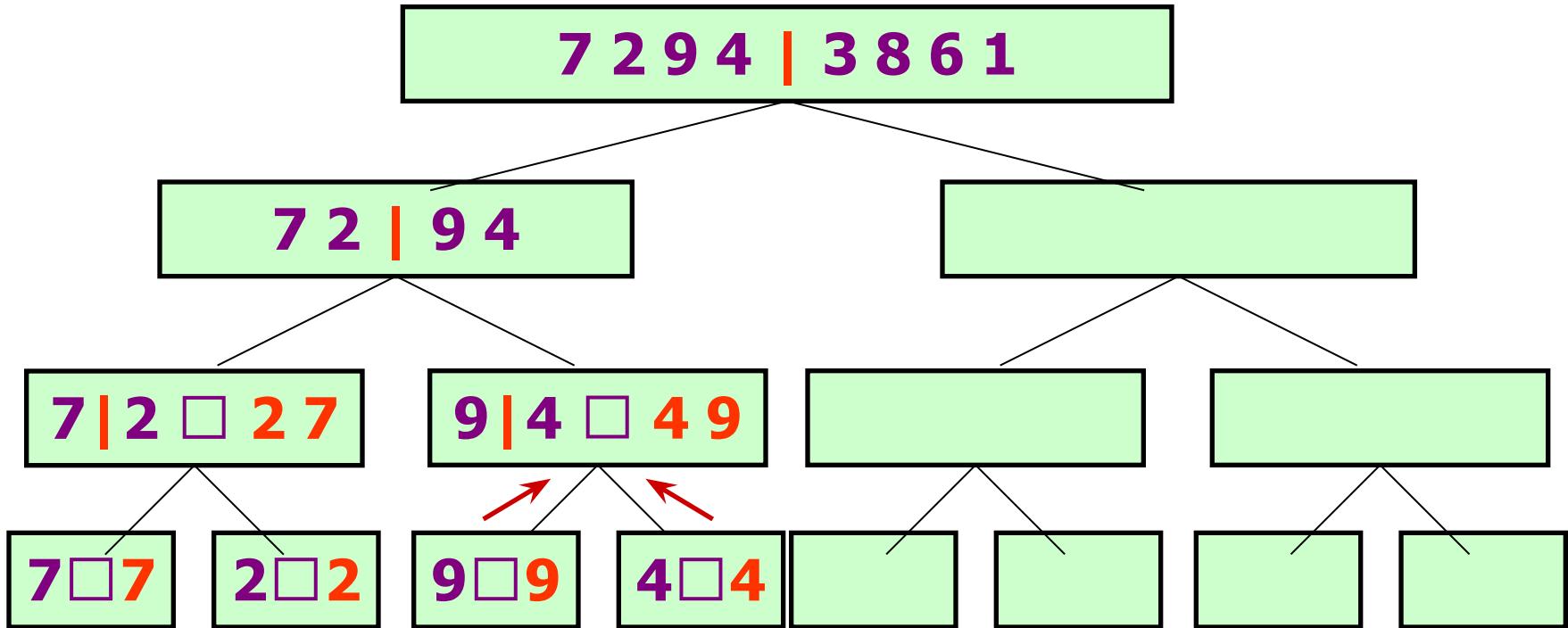
Merge



Merge Sort - example

Amity School of Engineering & Technology

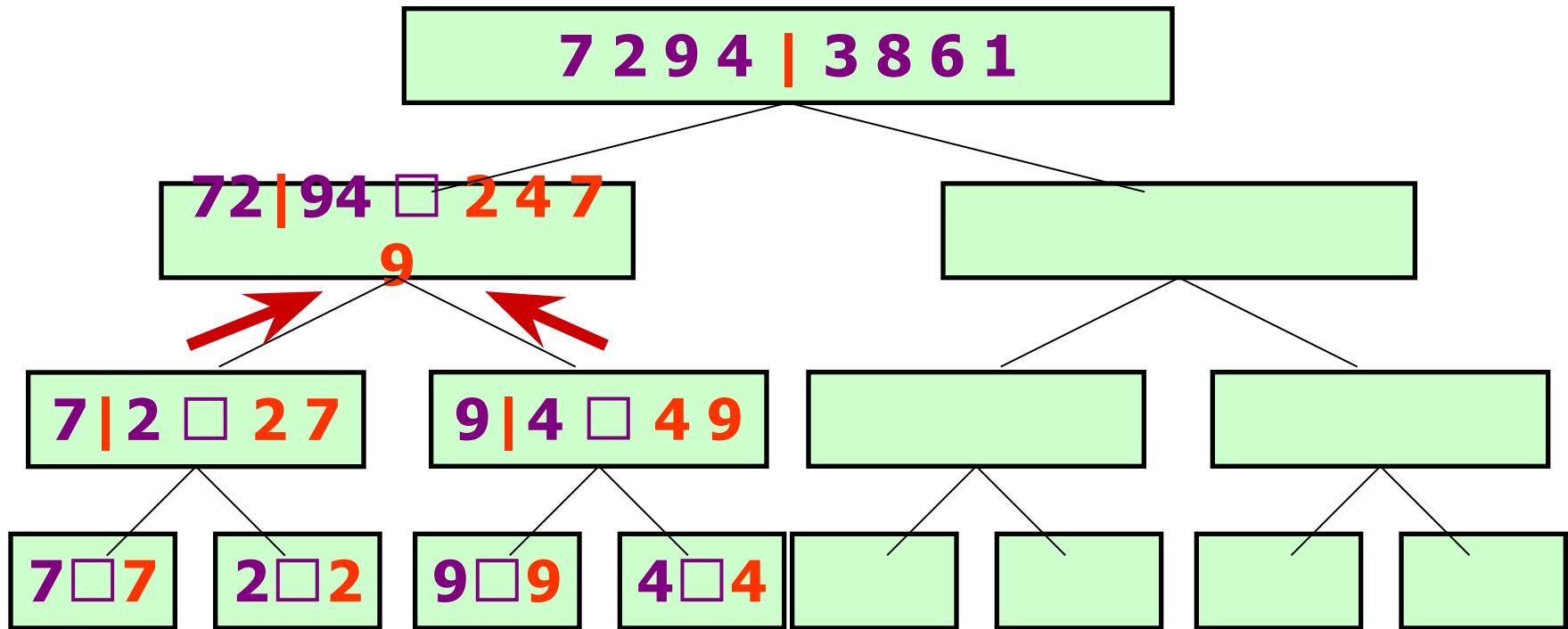
Recursive call, base case, ..., base case, merge



Merge Sort - example

Amity School of Engineering & Technology

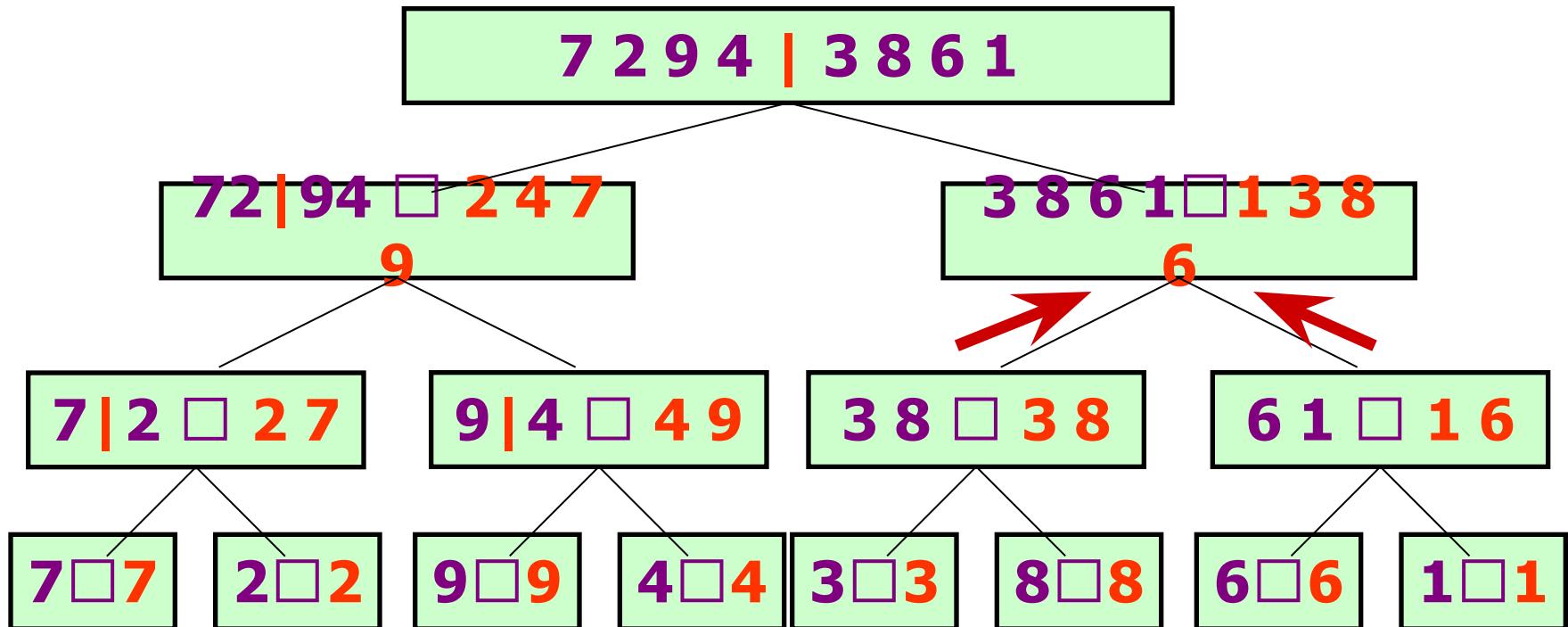
Merge



Merge Sort - example

Amity School of Engineering & Technology

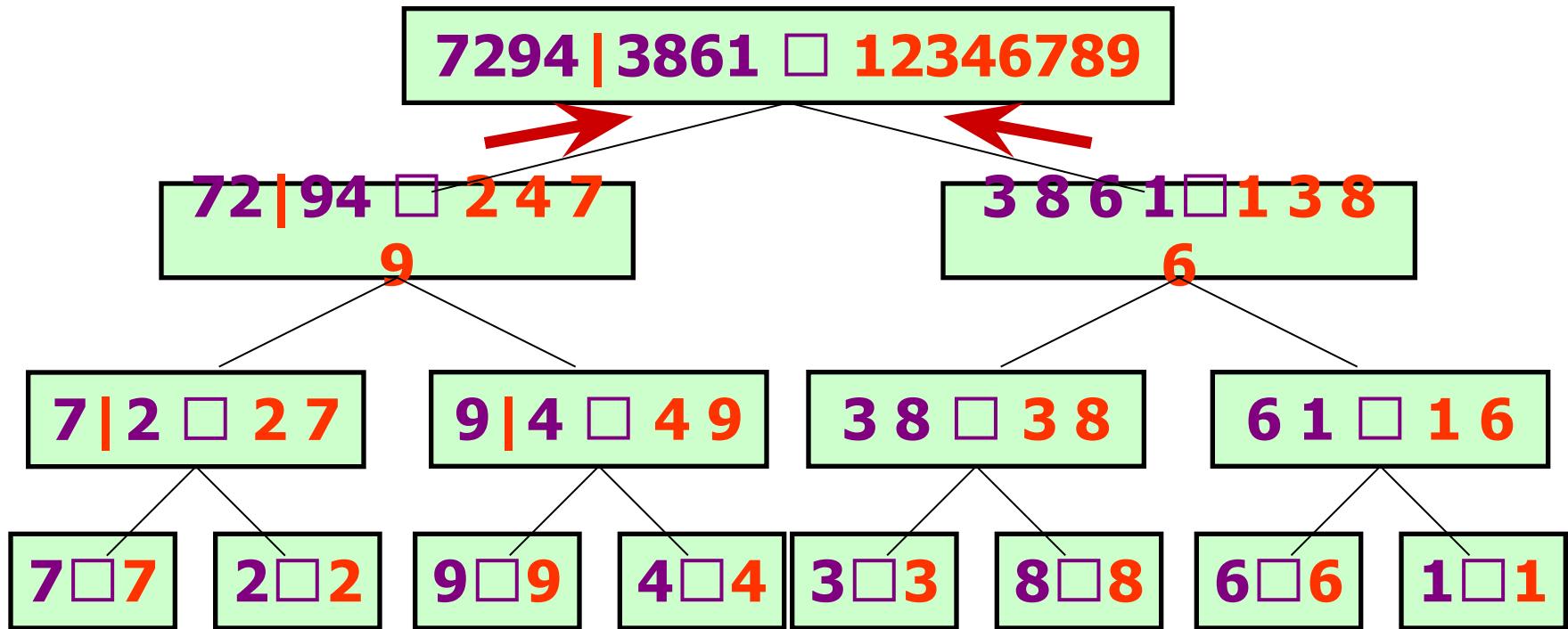
Recursive call, ..., merge, merge



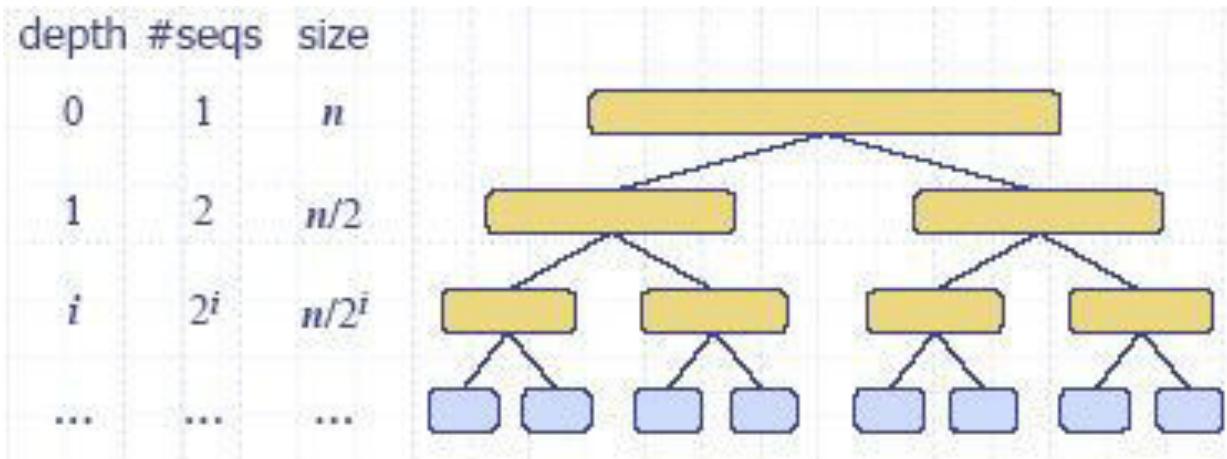
Merge Sort - example

Amity School of Engineering & Technology

Merge



- The height h of the merge-sort tree is $\log n + 1 = O(\log n)$
 - at each recursive call we divide in half the sequence
- The overall amount or work done at the nodes of depth i is $O(n)$
 - we partition and merge 2^i sequences of size $n/2^i$
 - we make 2^{i+1} recursive calls
- Thus, the total running time of merge-sort is $O(n \log n)$



```
void mergesort(int low, int
high)
{
    if (low<high)
    {
        int middle=(low+high)/2;
        mergesort(low, middle);
        mergesort(middle+1, high);
        merge(low, middle, high);
    }
}
```

Merge Sort – sample code (cont.)

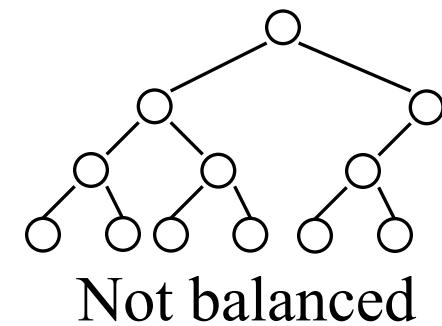
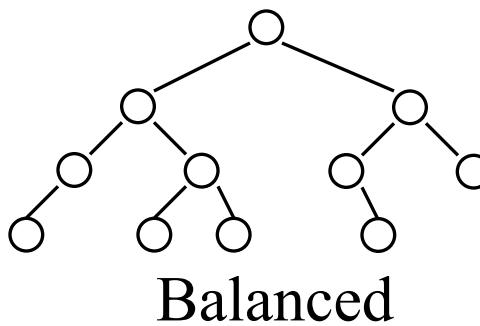
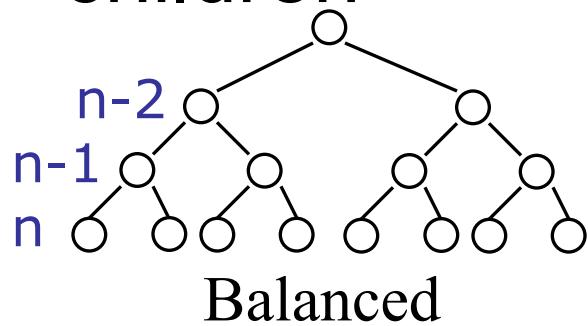
```
void merge(int low, int middle, int high)
{
    int i, j, k;
    // copy both halves of a to auxiliary array b
    for (i=low; i<=high; i++)
        b[i]=a[i];
    i=low; j=middle+1; k=low;
    // copy back next-greatest element at each time
    while (i<=middle && j<=high)
        if (b[i]<=b[j]) a[k++]=b[i++];
        else a[k++]=b[j++];
    // copy back remaining elements of first half (if
any)
    while (i<=middle)
        a[k++]=b[i++];
}
```

- Binary tree
- Balanced
- Left-justified or Complete
- (Max) Heap property: no node has a value greater than the value in its parent

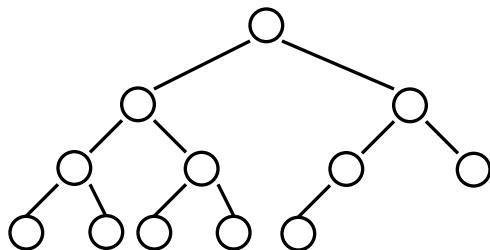
Balanced binary trees

Amity School of Engineering & Technology

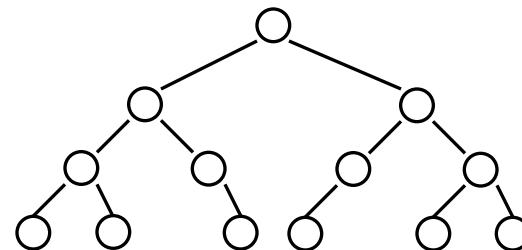
- Recall:
 - The depth of a node is its distance from the root
 - The depth of a tree is the depth of the deepest node
- A binary tree of depth n is balanced if all the nodes at depths 0 through $n-2$ have two children



- A balanced binary tree of depth n is left-justified if:
 - it has 2^n nodes at depth n (the tree is “full”), or
 - it has 2^k nodes at depth k , for all $k < n$, and all the leaves at depth n are as far left as possible



Left-justified



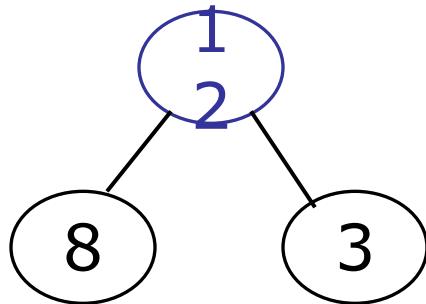
Not left-justified

- How to build a heap
- How to maintain a heap
- How to use a heap to sort data

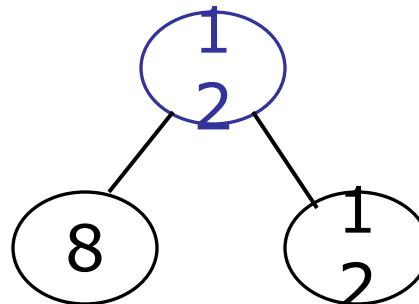
The heap property

Amity School of Engineering & Technology

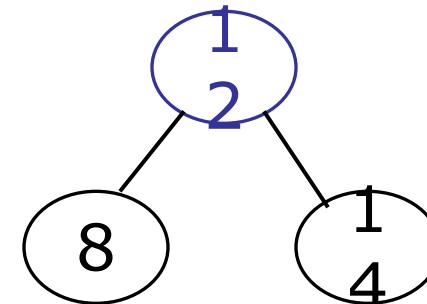
- A node has the heap property if the value in the node is as large as or larger than the values in its children



Blue node has
heap property



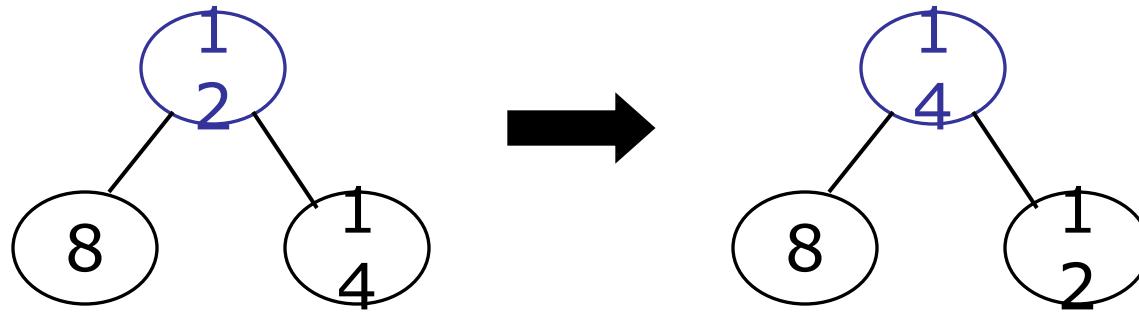
Blue node has
heap property



Blue node does not
have heap property

- All leaf nodes automatically have the heap property
- A binary tree is a heap if *all* nodes in it have the heap property

- Given a node that does not have the heap property, you can give it the heap property by exchanging its value with the value of the larger child

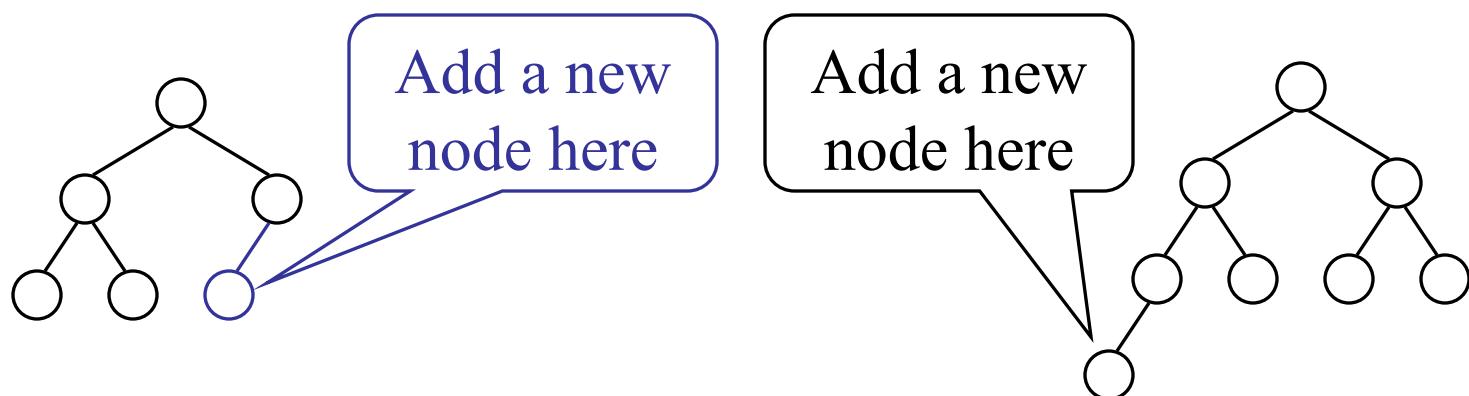


Blue node does not have heap property

Blue node has heap property

- This is sometimes called shifting up

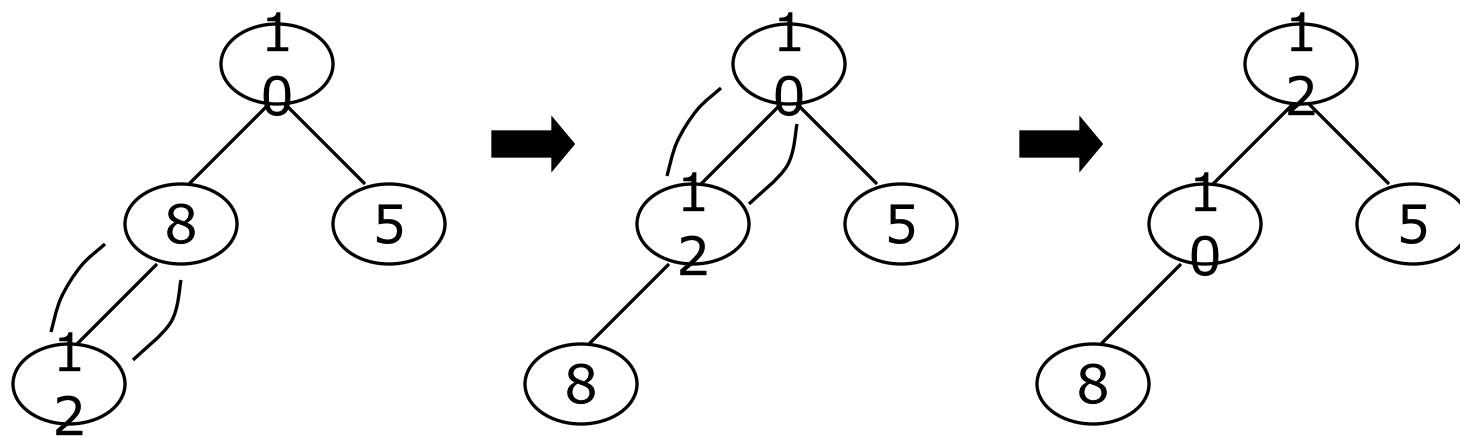
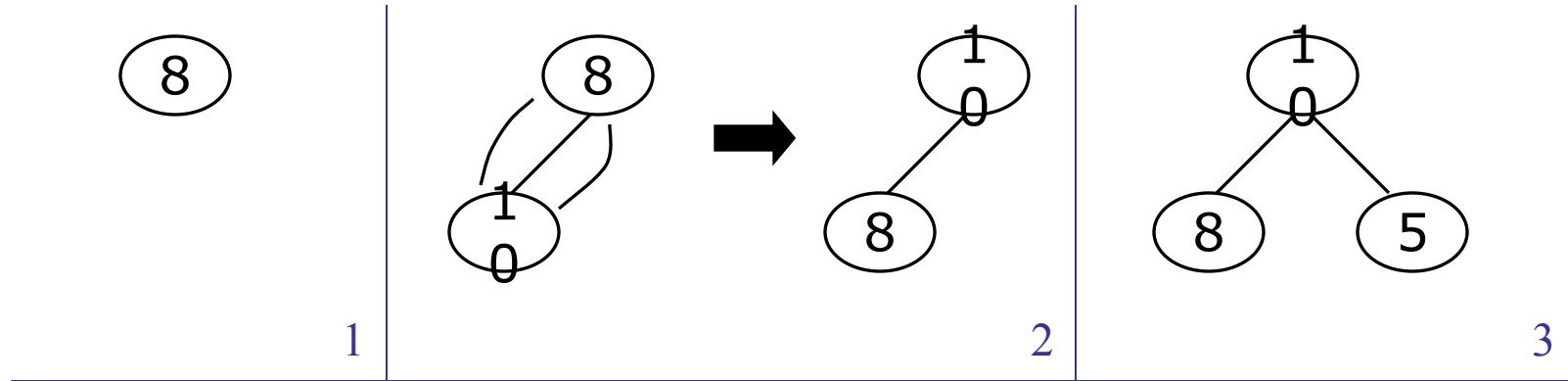
- A tree consisting of a single node is automatically a heap
- We construct a heap by adding nodes one at a time:
 - Add the node just to the right of the rightmost node in the deepest level
 - If the deepest level is full, start a new level
- Examples:



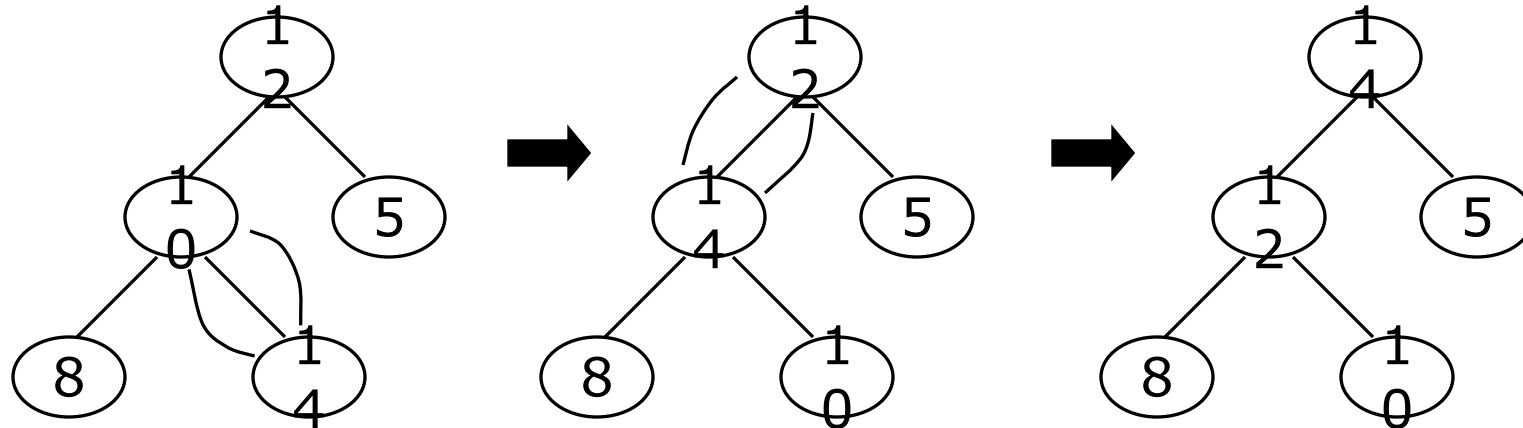
- Each time we add a node, we may destroy the heap property of its parent node
- To fix this, we sift up
- But each time we sift up, the value of the topmost node in the sift may increase, and this may destroy the heap property of *its* parent node
- We repeat the sifting up process, moving up in the tree, until either
 - We reach nodes whose values don't need to be swapped (because the parent is *still* larger than both children), or
 - We reach the root

Constructing a heap III

Amity School of Engineering & Technology



4

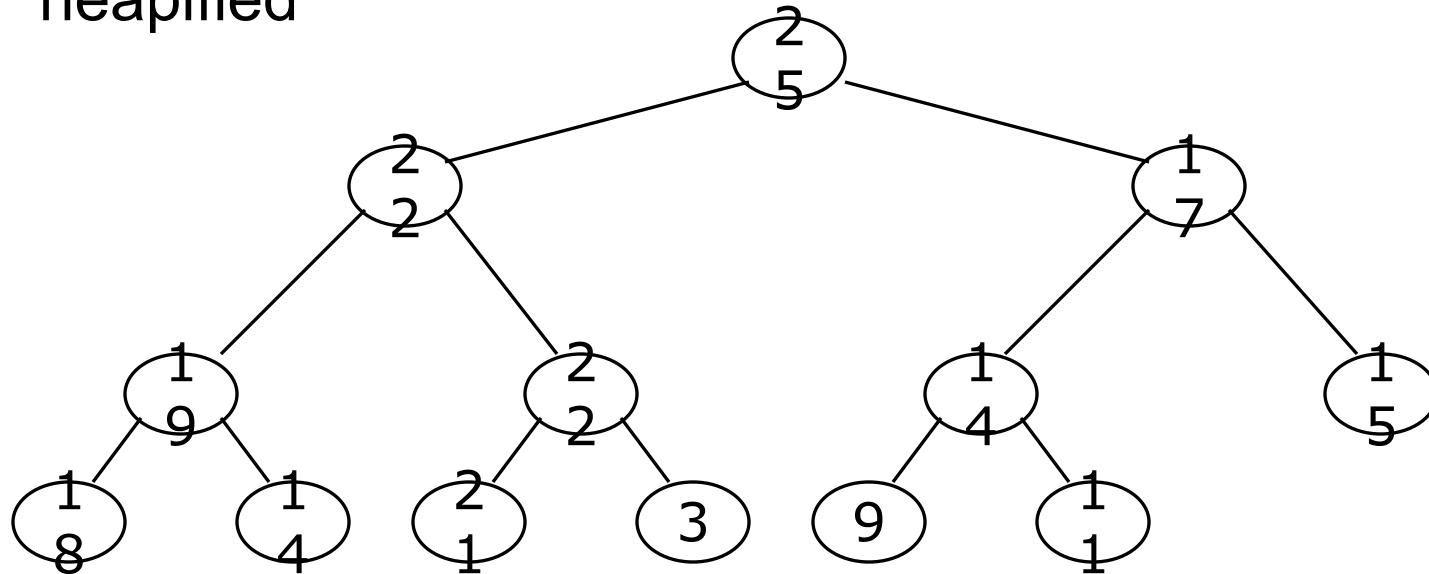


- The node containing 8 is not affected because its parent gets larger, not smaller
- The node containing 5 is not affected because its parent gets larger, not smaller
- The node containing 8 is still not affected because, although its parent got smaller, its parent is still greater than it was originally

A sample heap

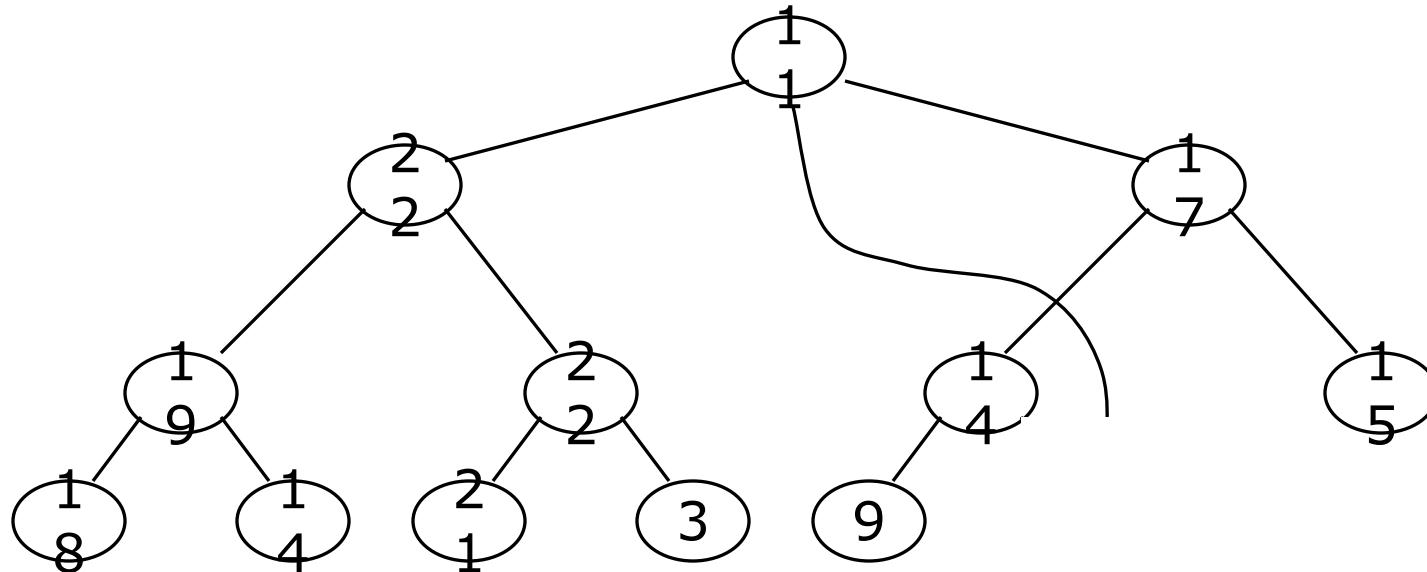
Amity School of Engineering & Technology

- Here's a sample binary tree after it has been heapified



- Notice that heapified does *not* mean sorted
- Heapifying does *not* change the shape of the binary tree; this binary tree is balanced and left-justified because it started out that way

- Notice that the largest number is now in the root
- Suppose we *discard* the root:

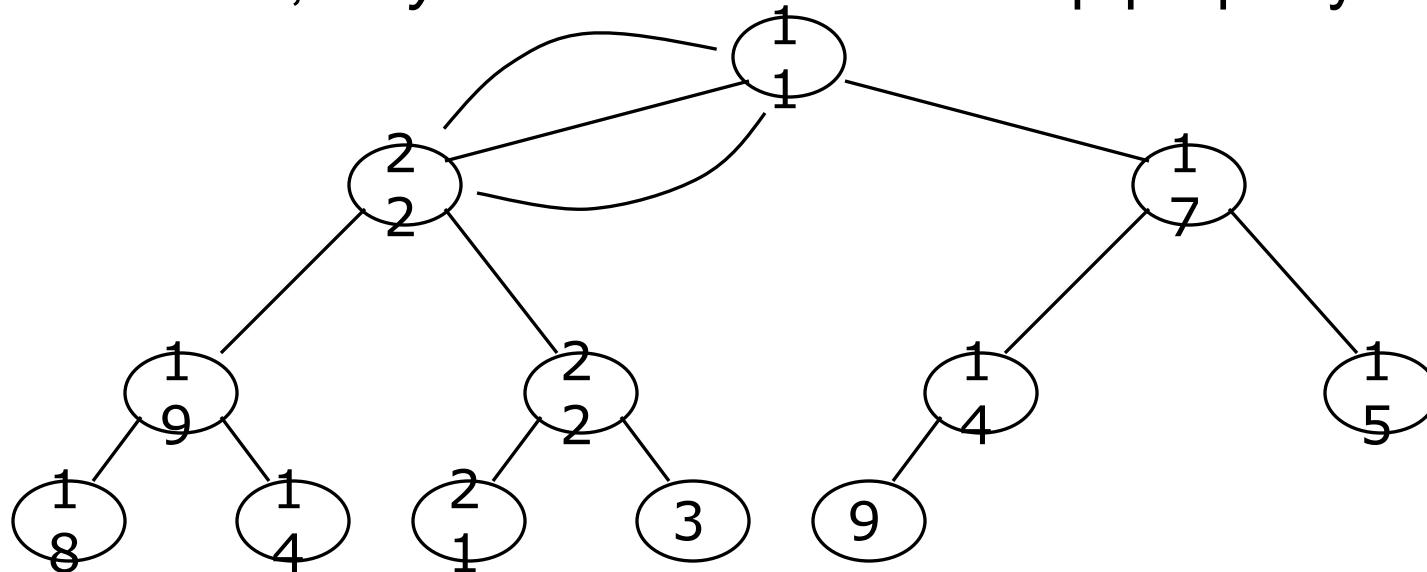


- How can we fix the binary tree so it is once again *balanced and left-justified*?
- Solution: remove the rightmost leaf at the deepest level and use it for the new root

The reHeap method I

Amity School of Engineering & Technology

- Our tree is balanced and left-justified, but no longer a heap
- However, *only the root* lacks the heap property

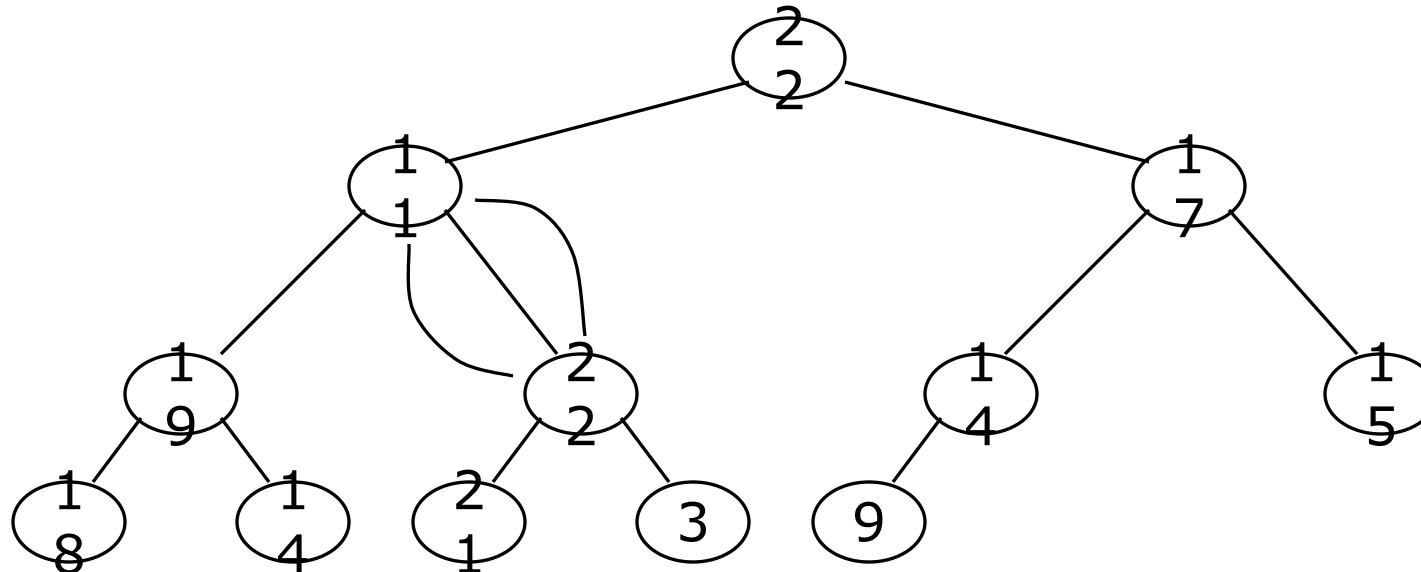


- We can `shiftDown()` the root
- After doing this, one and only one of its children may have lost the heap property

The reHeap method II

Amity School of Engineering & Technology

- Now the left child of the root (still the number 11) lacks the heap property

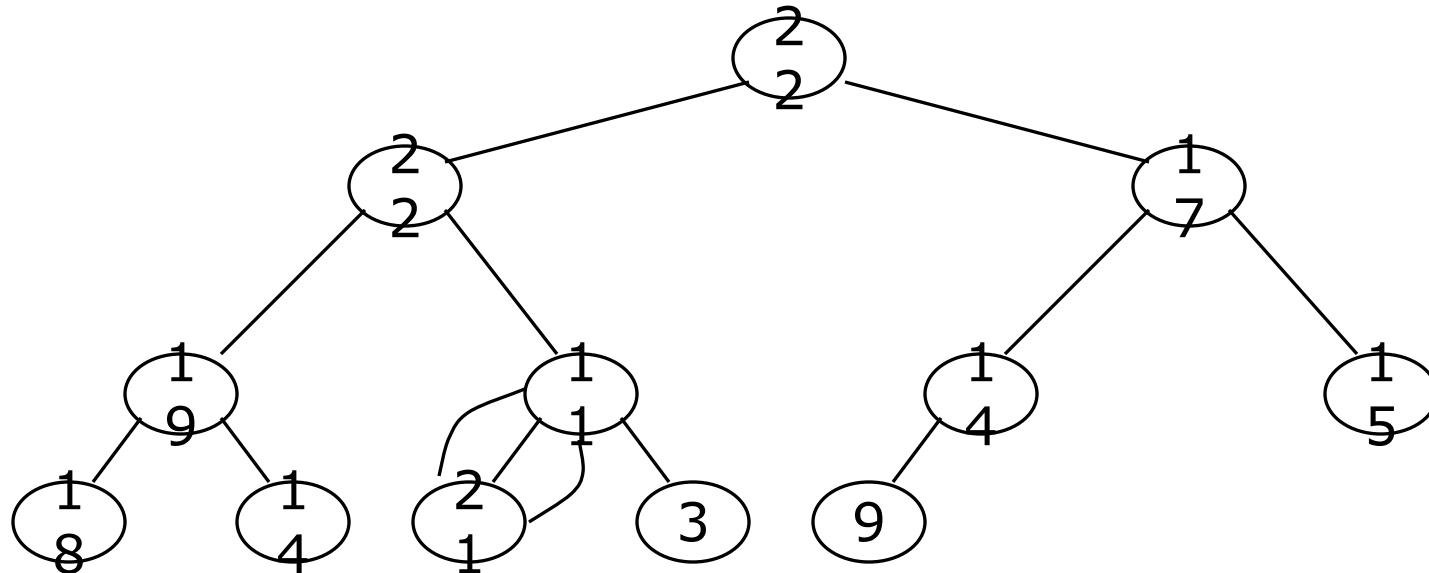


- We can `siftDown()` this node
- After doing this, one and only one of its children may have lost the heap property

The reHeap method III

Amity School of Engineering & Technology

- Now the right child of the left child of the root (still the number 11) lacks the heap property:

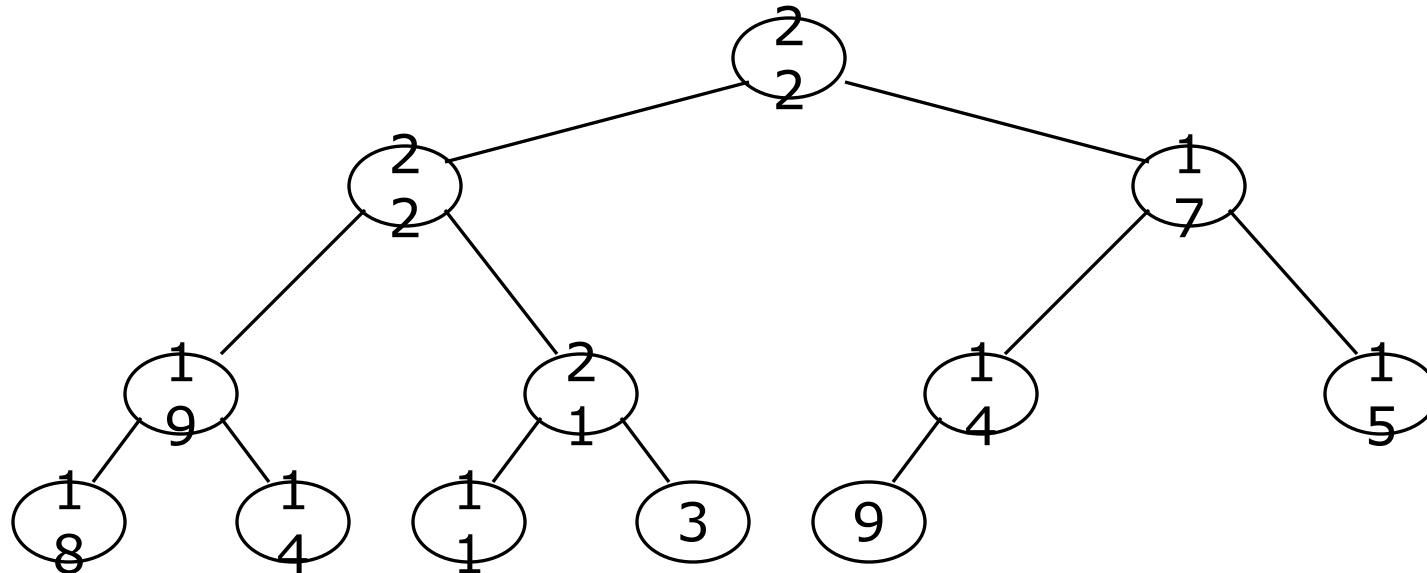


- We can `shiftDown()` this node
- After doing this, one and only one of its children may have lost the heap property —but it doesn't, because it's a leaf

The reHeap method IV

Amity School of Engineering & Technology

- Our tree is once again a heap, because every node in it has the heap property



- Once again, the largest (or a largest) value is in the root
- We can repeat this process until the tree becomes empty
- This produces a sequence of values in order largest to smallest

- What do heaps have to do with sorting an array?
- Here's the neat part:
 - Because the binary tree is *balanced* and *left justified*, it can be represented as an array
 - All our operations on binary trees can be represented as operations on arrays
 - To sort:

```
heapify the array;  
while the array isn't empty {  
    remove and replace the root;  
    reheap the new root node;  
}
```

Key properties

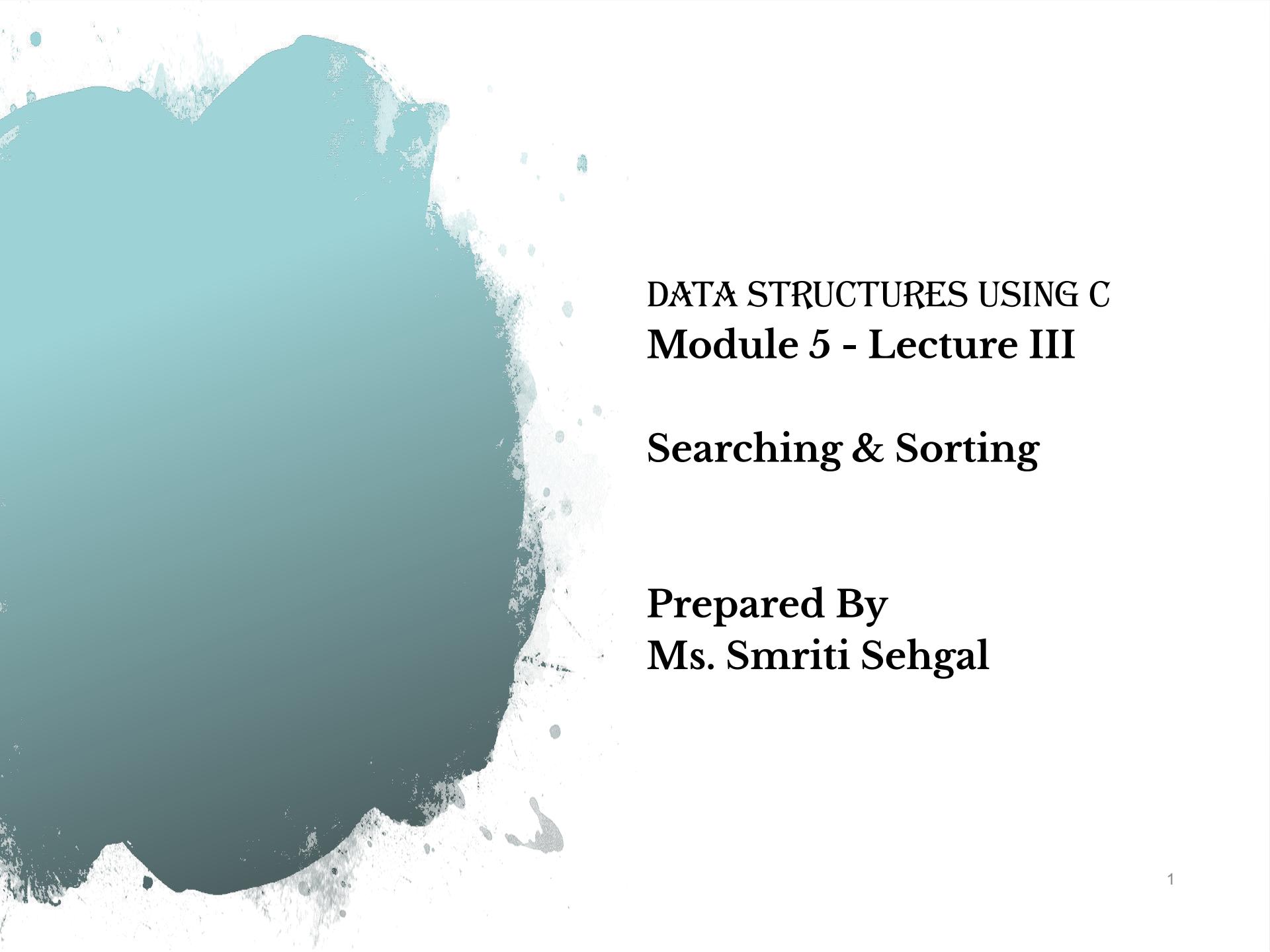
Amity School of Engineering & Technology

- Determining location of root and “last node” take constant time
- Remove n elements, re-heap each time

- To reheap the root node, we have to follow *one path* from the root to a leaf node (and we might stop before we reach a leaf)
- The binary tree is perfectly balanced
- Therefore, this path is $O(\log n)$ long
 - And we only do $O(1)$ operations at each node
 - Therefore, reheap takes $O(\log n)$ times
- Since we reheap inside a while loop that we do n times, the total time for the while loop is $n * O(\log n)$, or $O(n \log n)$

- Construct the heap $O(n \log n)$
- Remove and re-heap $O(\log n)$
 - Do this n times $O(n \log n)$
- Total time $O(n \log n) + O(n \log n)$





DATA STRUCTURES USING C

Module 5 - Lecture III

Searching & Sorting

Prepared By
Ms. Smriti Sehgal

Syllabus

Insertion Sort, Bubble sort, Selection sort, Quick sort, Merge sort, Heap sort, **Partition exchange sort, Shell sort, Sorting on different keys, External sorting.** Linear search, Binary search, Hashing: Hash Functions, Collision Resolution Techniques.

In this session

We will talk about

- **Partition exchange sort**
- **Shell sort**
- **Sorting on different keys**
- **External sorting.**

Partition Exchange Sort

- **Quicksort** (sometimes called **partition-exchange sort**) is an efficient sorting algorithm, serving as a systematic method for placing the elements of a random access file or an array in order.

Shell Sort

- Shell sort is a highly efficient sorting algorithm and is based on insertion sort algorithm.
- This algorithm avoids large shifts as in case of insertion sort, if the smaller value is to the far right and has to be moved to the far left.
- This algorithm uses insertion sort on a widely spread elements, first to sort them and then sorts the less widely spaced elements. This spacing is termed as **interval**. This interval is calculated based on Knuth's formula as –

Knuth's Formula

$$h = h * 3 + 1 \text{ where } - h \text{ is interval with initial value}$$

Shell Sort

Amity School of Engineering & Technology

- This algorithm is quite efficient for medium-sized data sets as its average and worst-case complexity of this algorithm depends on the gap sequence the best known is $O(n)$, where n is the number of items.
- And the worst case space complexity is $O(n)$.

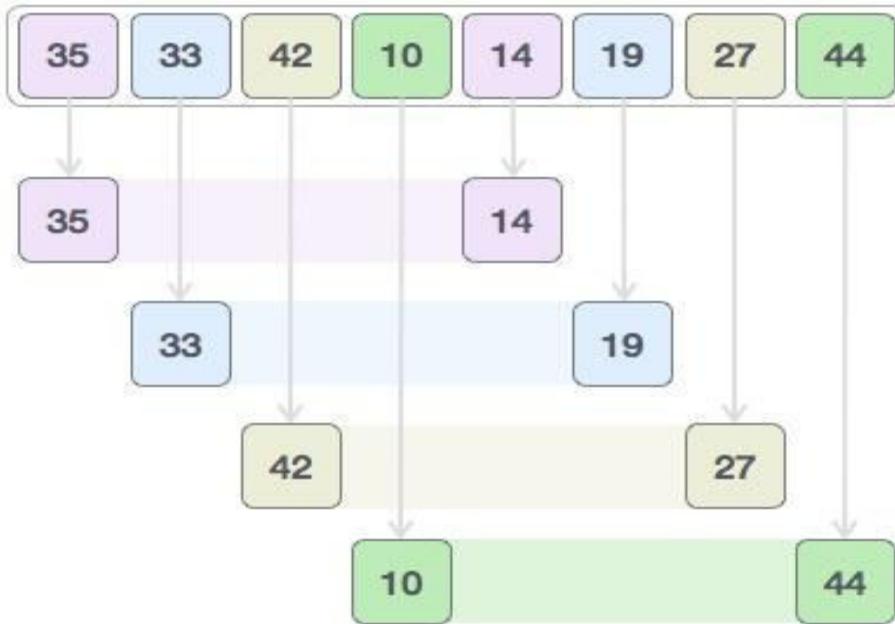
How shell sort works?

Amity School of Engineering & Technology

- Let us consider the following example to have an idea of how shell sort works.
- For our example and ease of understanding, we take the interval of 4.
- Make a virtual sub-list of all values located at the interval of 4 positions. Here these values are {35, 14}, {33, 19}, {42, 27} and {10, 44}

How shell sort works?

Amity School of Engineering & Technology



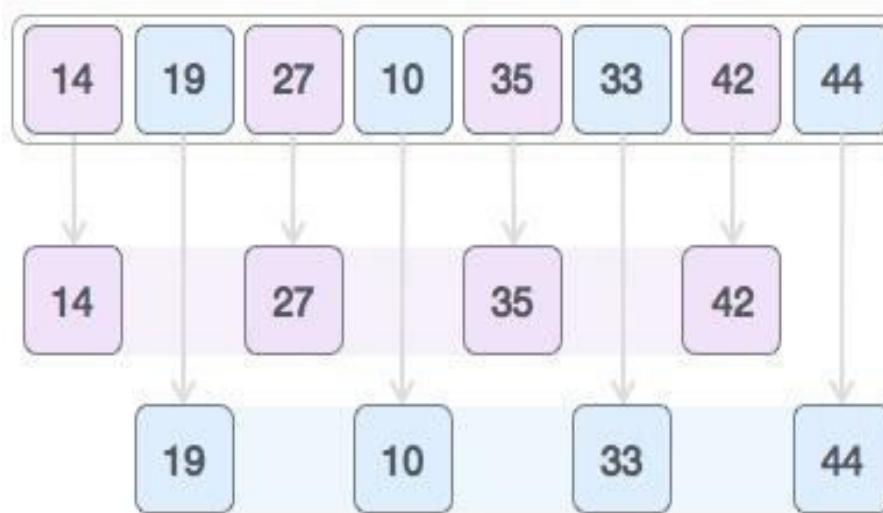
- We compare values in each sub-list and swap them (if necessary) in the original array. After this step, the new array should look like this –



How shell sort works?

Amity School of Engineering & Technology

- Then, we take interval of 1 and this gap generates two sub-lists - {14, 27, 35, 42}, {19, 10, 33, 44}



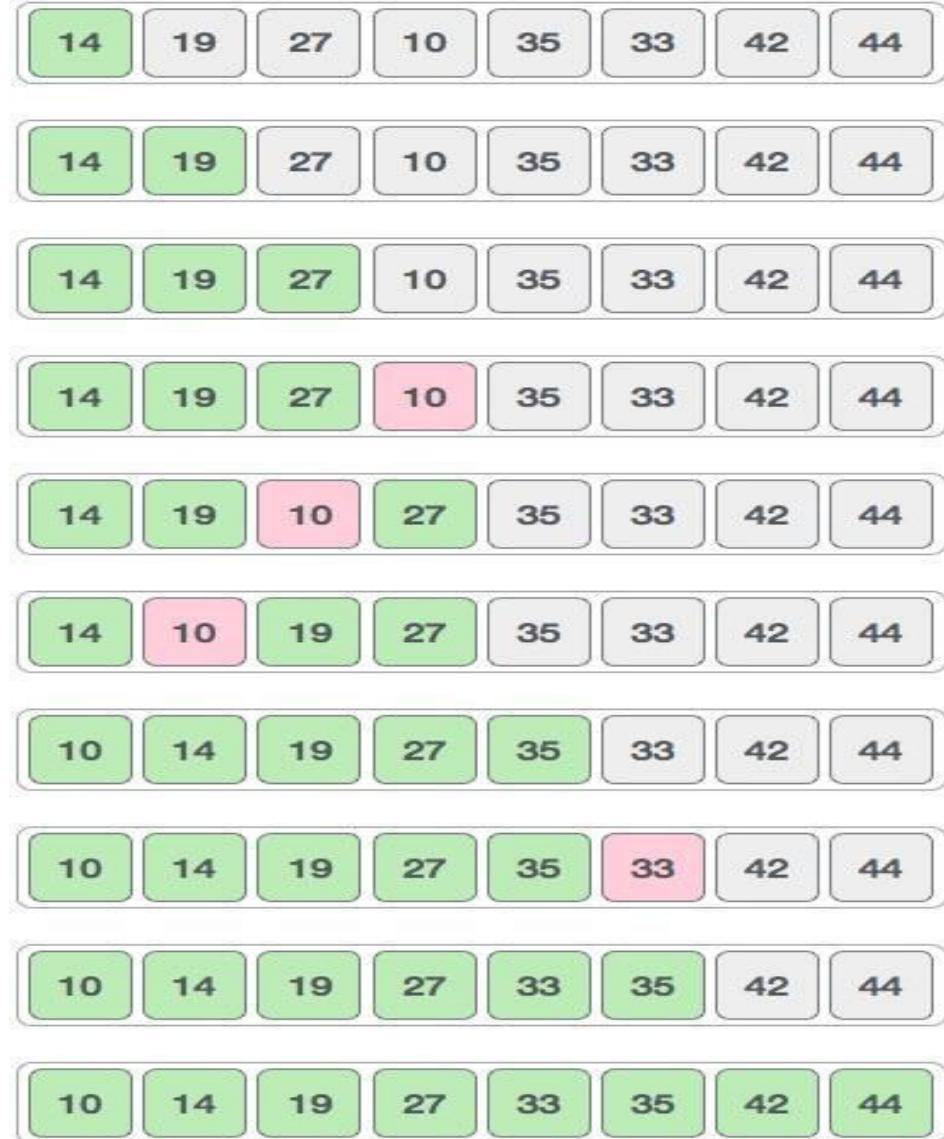
- We compare and swap the values, if required, in the original array. After this step, the array should look like this –



How shell sort works?

Amity School of Engineering & Technology

- Finally, we sort the rest of the array using interval of value 1. Shell sort uses insertion sort to sort the array.
- Following is the step-by-step depiction
- We see that it required only four swaps to sort the rest of the array.



Algorithm

Following is the algorithm for shell sort.

Step 1 – Initialize the value of h

Step 2 – Divide the list into smaller sub-list of equal interval h

Step 3 – Sort these sub-lists using **insertion sort**

Step 4 – Repeat until complete list is sorted

Key points of shell sort algorithm

- Shell Sort is a comparison based sorting.
- Time complexity of Shell Sort depends on gap sequence. Its best case time complexity is $O(n^* \log n)$ and worst case is $O(n^* \log^2 n)$. Time complexity of Shell sort is generally assumed to be near to $O(n)$ and less than $O(n^2)$ as determining its time complexity is still an open problem.
- The best case in shell sort is when the array is already sorted. The number of comparisons is less.
- It is an in-place sorting algorithm as it requires no additional scratch space.
- Shell Sort is unstable sort as relative order of elements with equal values may change.
- It is been observed that shell sort is 5 times faster than bubble sort and twice faster than insertion sort its closest competitor.
- There are various increment sequences or gap sequences in shell sort which produce various complexity between $O(n)$ and $O(n^2)$.

Sort on different keys

Amity School of Engineering & Technology

- Sorting refers to the operation or technique of arranging and rearranging sets of data in some specific order. A collection of records called a list where every record has one or more fields. The fields which contain a unique value for each record is termed as the **key** field.
- For example, a phone number directory can be thought of as a list where each record has three fields - 'name' of the person, 'address' of that person, and their 'phone numbers'. Being unique phone number can work as a key to locate any record in the list. Sorting is the operation performed to arrange the records of a table or list in some order according to some specific ordering criterion. Sorting is performed according to some key value of each record. The records are either sorted either numerically or alphanumerically. The records are then arranged in ascending or descending order depending on the numerical value of the key. Here is an example, where the sorting of a lists of marks obtained by a student in any particular subject of a class.

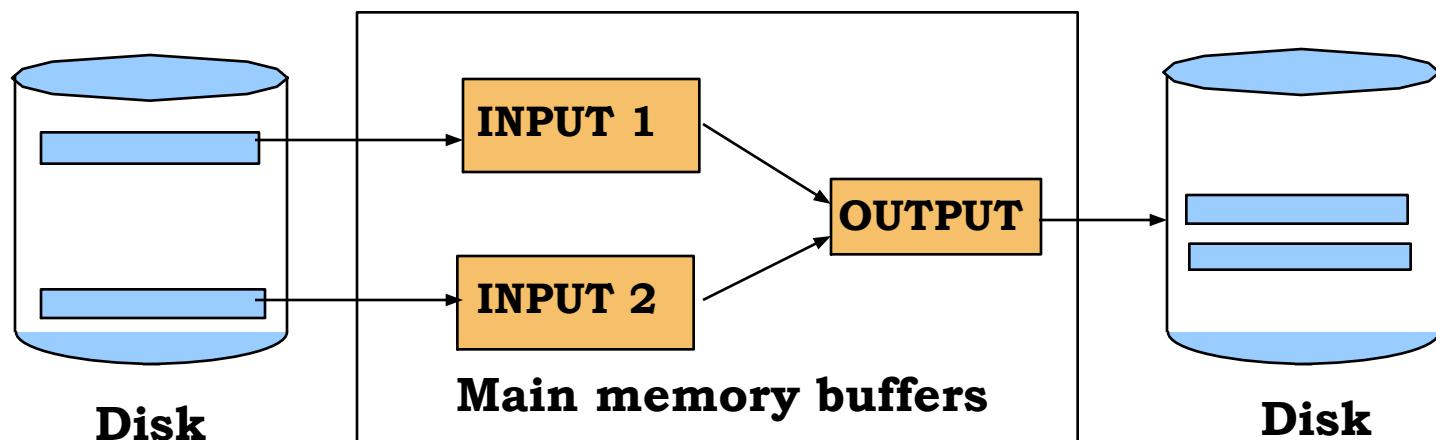
Why Sort?

Amity School of Engineering & Technology

- A classic problem in computer science!
- Data requested in sorted order
 - e.g., find students in increasing gpa order
- Sorting is first step in *bulk loading* B+ tree index.
- Sorting useful for eliminating *duplicate copies* in a collection of records (Why?)
- *Sort-merge* join algorithm involves sorting.
- Problem: sort 1Gb of data with 1Mb of RAM.
 - why not virtual memory?

2-Way Sort: Requires 3 Buffers

- Pass 1: Read a page, sort it, write it.
 - only one buffer page is used
- Pass 2, 3, ..., etc.:
 - three buffer pages used.



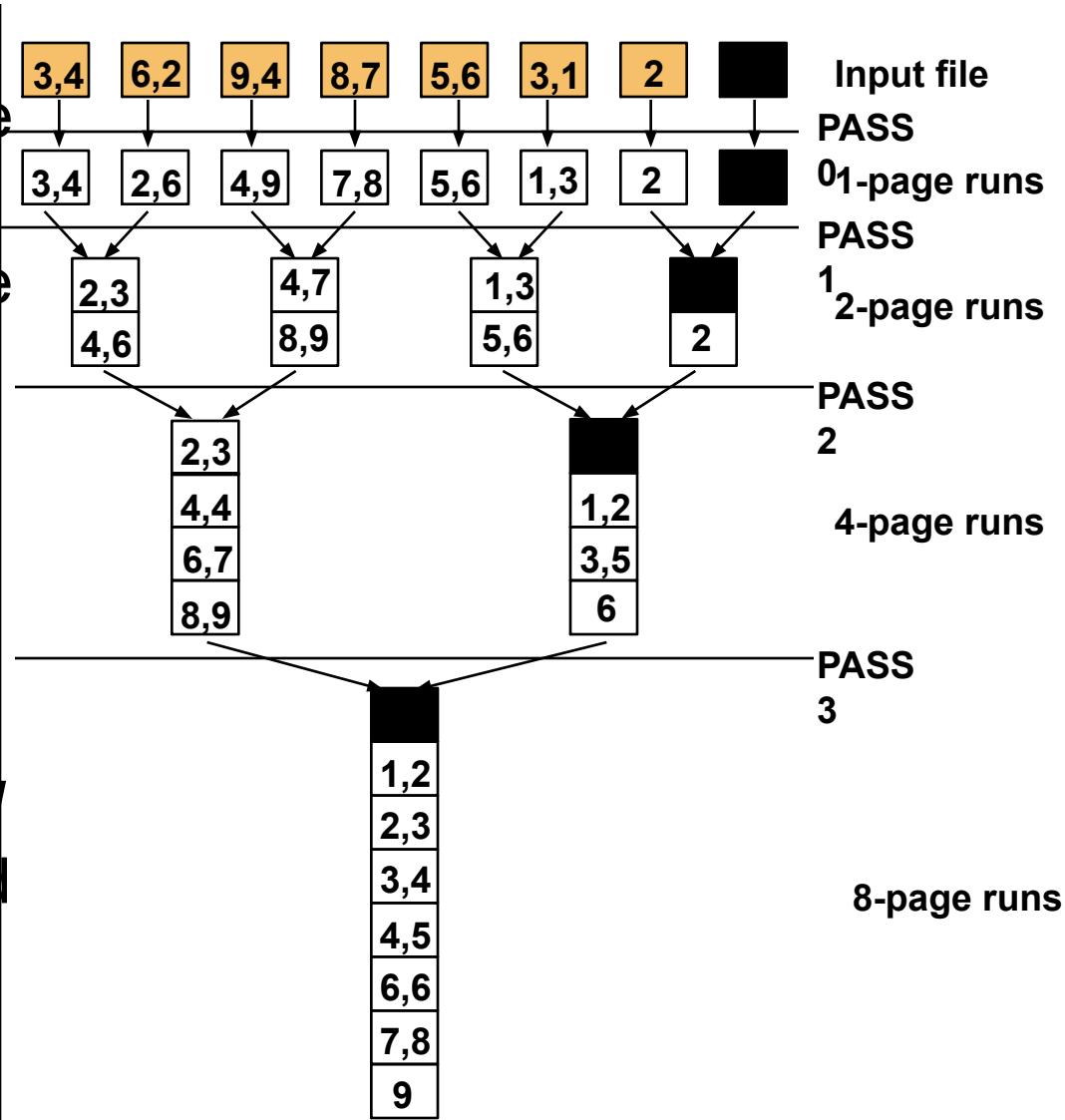
Two-Way External Merge Sort

Amity School of Engineering & Technology

- Each pass we read + write each page in file.
- N pages in the file \Rightarrow the number of passes
 $= \lceil \log_2 N \rceil + 1$
- So total cost is:

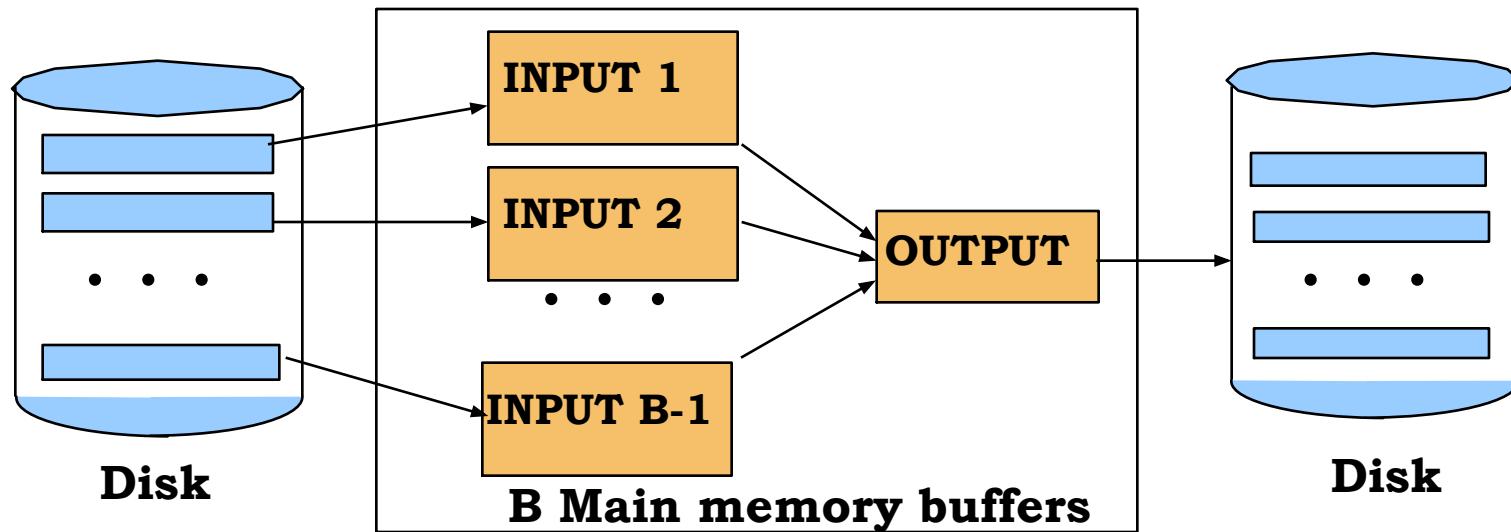
$$2N(\lceil \log_2 N \rceil + 1)$$

- Idea: **Divide and conquer:** sort subfiles and merge



• More than 3 buffer pages. How can we utilize them?

- To sort a file with N pages using B buffer pages:
 - Pass 0: use B buffer pages. Produce $\lceil N / B \rceil$ sorted runs of B pages each.
 - Pass 2, ..., etc.: merge $B-1$ runs.



- Number of passes: $1 + \lceil \log_{B-1} \lceil N / B \rceil \rceil$
- Cost = $2N * (\# \text{ of passes})$
- E.g., with 5 buffer pages, to sort 108 page file:
 - Pass 0: $\lceil 108 / 5 \rceil = 22$ sorted runs of 5 pages each (last run is only 3 pages)
 - Pass 1: $\lceil 22 / 4 \rceil = 6$ sorted runs of 20 pages each (last run is only 8 pages)
 - Pass 2: 2 sorted runs, 80 pages and 28 pages
 - Pass 3: Sorted file of 108 pages

Number of Passes of External Sort

Amity School of Engineering & Technology

N	B=3	B=5	B=9	B=17	B=129	B=257
100	7	4	3	2	1	1
1,000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

- ... longer runs often means fewer passes!
- Actually, do I/O a page at a time
- In fact, read a *block* of pages sequentially!
- Suggests we should make each buffer (input/output) be a *block* of pages.
 - But this will reduce fan-out during merge passes!
 - In practice, most files still sorted in 2-3 passes.

Number of Passes of Optimized Sort

Amity School of Engineering & Technology

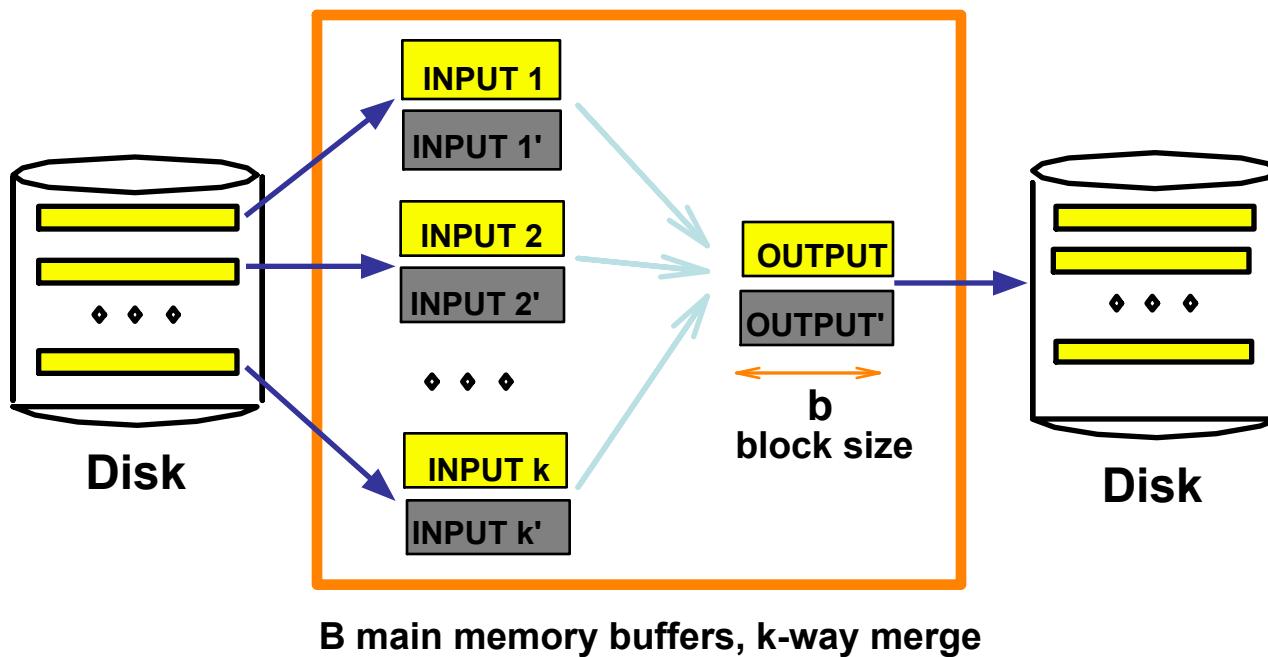
N	B=1,000	B=5,000	B=10,000
100	1	1	1
1,000	1	1	1
10,000	2	2	1
100,000	3	2	2
1,000,000	3	2	2
10,000,000	4	3	3
100,000,000	5	3	3
1,000,000,000	5	4	3

- Block size = 32, initial pass produces runs of size $2B$.

Double Buffering

Amity School of Engineering & Technology

- To reduce wait time for I/O request to complete, can *prefetch* into 'shadow block'.
 - Potentially, more passes; in practice, most files still sorted in **2-3 passes**.



Sorting Records!

Amity School of Engineering & Technology

- Sorting has become a blood sport!
 - Parallel sorting is the name of the game ...
- Datamation: Sort 1M records of size 100 bytes
 - Typical DBMS: 15 minutes
 - World record: 3.5 **seconds**
 - 12-CPU SGI machine, 96 disks, 2GB of RAM
- New benchmarks proposed:
 - Minute Sort: How many can you sort in 1 minute?
 - Dollar Sort: How many can you sort for \$1.00?



Amity School of Engineering & Technology

Programme: B.Tech, Semester- III

Data Structure Using C

Module-V

Searching & Sorting Techniques

- Insertion Sort, Bubble sort, Selection sort, Quick sort, Merge sort, Heap sort, Partition exchange sort, Shell sort, Sorting on different keys, External sorting. **Linear search, Binary search, Hashing:** Hash Functions, Collision Resolution Techniques.

In this session

We will talk about

- Linear search
- Binary search

What is algorithm and Algorithm design?

- An Algorithm is a Step by Step solution of a specific mathematical or computer related problem.
- Algorithm design is a specific method to create a mathematical process in solving problems.

Sorted Array

- Sorted array is an array where each element is **sorted** in numerical, alphabetical, or some other order, and placed at equally spaced addresses in computer memory.

0	1	2	3
10	20	30	40

Unsorted Array

- Unsorted array is an array where each element is not **sorted** in numerical, alphabetical, or some other order, and placed at equally spaced addresses in computer memory.

0	1	2	3
10	40	30	20

What is Searching?

- In computer science, searching is the process of finding an item with specified properties from a collection of items.
- The items may be stored as records in a database, simple data elements in arrays, text in files, nodes in trees, vertices and edges in graphs, or maybe be elements in other search place.
- The **definition** of a **search** is the process of looking for something or someone
- Example: An example of a **search** is a quest to find a missing person

Why do we need searching?

- Searching is one of the core computer science algorithms.
- We know that today's computers store a lot of information.
- To retrieve this information proficiently we need very efficient searching algorithms.
- **Types of Searching**
 - Linear search
 - Binary search

Linear search

- The linear search is a sequential search, which uses a loop to step through an array, starting with the first element.
- It compares each element with the value being searched for, and stops when either the value is found or the end of the array is encountered.
- If the value being searched is not in the array, the algorithm will unsuccessfully search to the end of the array.

Linear search

- Since the array elements are stored in linear order searching the element in the linear order make it easy and efficient.
- The search may be successful or unsuccessfully. That is, if the required element is found them the search is successful other wise it is unsuccessfully.

Linear search

- Since the array elements are stored in linear order searching the element in the linear order make it easy and efficient.
- The search may be successful or unsuccessfully. That is, if the required element is found them the search is successful other wise it is unsuccessfully.

Unordered linear/ Sequential search

```
int unorderedlinearsearch (int A[], int n, int data)
{
    for (int i=0; i<n; i++)
    {
        if(A[i] == data)
            return i;
    }
    return -1;
}
```

Advantages of Linear search

- If the first number in the directory is the number you were searching for ,then lucky you!!.
- Since you have found it on the very first page, now its not important for you that how many pages are there in the directory.
- The linear search is simple -It is very easy to understand and implement
- It does not require the data in the array to be stored in any particular order
- So it does not depends on no. on elements in the directory. Hence constant time .

Disadvantages of Linear search

- It may happen that the number you are searching for is the last number of directory or if it is not in the directory at all.
- In that case you have to search the whole directory.
- Now number of elements will matter to you. if there are 500 pages ,you have to search 500;if it has 1000 you have to search 1000.
- Your search time is proportional to number of elements in the directory.
- It has very poor efficiency because it takes lots of comparisons to find a particular record in big files
- The performance of the algorithm scales linearly with the size of the input
- Linear search is slower then other searching algorithms

Analysis of Linear search

- In the **average case**, the target value is somewhere in the array.
- In fact, since the target value can be anywhere in the array, any element of the array is equally likely.
- So on average, the target value will be in the middle of the array.
- So the search takes an amount of time proportional to half the length of the array
- The worst case complexity is $O(n)$, sometimes known an $O(n)$ search
- Time taken to search elements keep increasing as the number of elements are increased.

Analysis of Linear search

- How long will our search take?

In the **best case**, the target value is in the first element of the array.

So the search takes some tiny, and constant, amount of time.

In the **worst case**, the target value is in the last element of the array.

So the search takes an amount of time proportional to the length of the array.

Binary search

- The general term for a smart search through sorted data is a ***binary search***.

Step 1: The initial search region is the whole array.

Step 2: Look at the data value in the middle of the search region.

Step 3: If you've found your target, stop.

Step 4: If your target is less than the middle data value, the new search region is the lower half of the data.

Step 5: If your target is greater than the middle data value, the new search region is the higher half of the data.

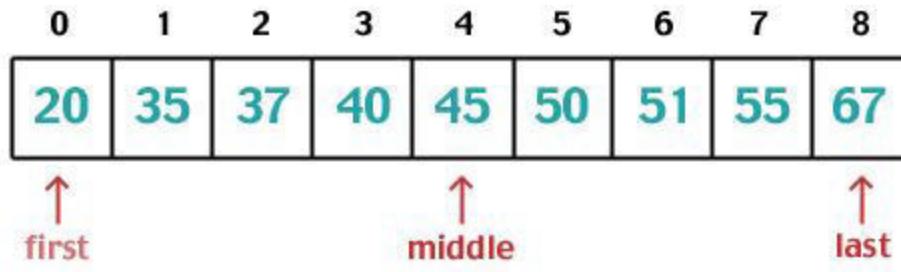
Step 6: Continue from Step 2.

Binary search

0	1	2	3	4	5	6	7	8
20	35	37	40	45	50	51	55	67

Binary search

Step2: Calculate middle = (low + high) / 2.
= $(0 + 8) / 2 = 4.$



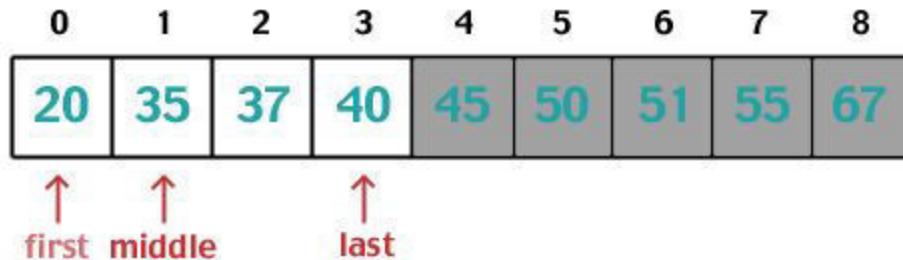
If $37 == \text{array}[middle]$ return middle

Else if $37 < \text{array}[middle]$ high = middle - 1

Else if $37 > \text{array}[middle]$ low = middle + 1

Binary search

Repeat Step 2: Calculate middle = (low + high) / 2.
 $= (0 + 3) / 2 = 1.$



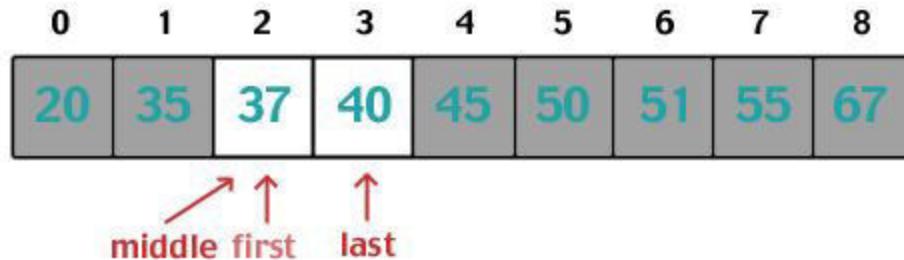
If 37 == array[middle] return middle

Else if 37 < array[middle] high = middle - 1

Else if 37 > array[middle] low = middle + 1

Binary search

Repeat Step 2: Calculate middle = (low + high) / 2.
 $= (2 + 3) / 2 = 2.$



If 37 == array[middle] return middle

Else if 37 < array[middle] high = middle -1

Else if 37 > array[middle] low = middle +1

Binary search Routine

```
public int binarySearch (int[] number, int
searchValue)
{
    int low = 0, high = number.length - 1, mid = (low +
high) / 2;

    while (low <= high && number[mid] != searchValue) {
        if (number[mid] < searchValue) {
            low = mid + 1;
        }
        else
            { //number[mid] > searchValue
                high = mid - 1;
            }
        mid = (low + high) / 2; //integer
division will truncate
    }

    if (low > high) {
        mid = NOT_FOUND;
    }
    return mid;
}
```

Binary Search Performance

- Successful Search
 - Best Case –1 comparison
 - Worst Case – $\log_2 N$ comparisons
- Unsuccessful Search
 - Best Case = ----
 - Worst Case = $\log_2 N$ comparisons
- Since the portion of an array to search is cut into half after every comparison, we compute how many times the array can be divided into halves.
- After K comparisons, there will be $N/2^K$ elements in the list. We solve for K when $N/2^K= 1$, deriving $K = \log_2 N$.

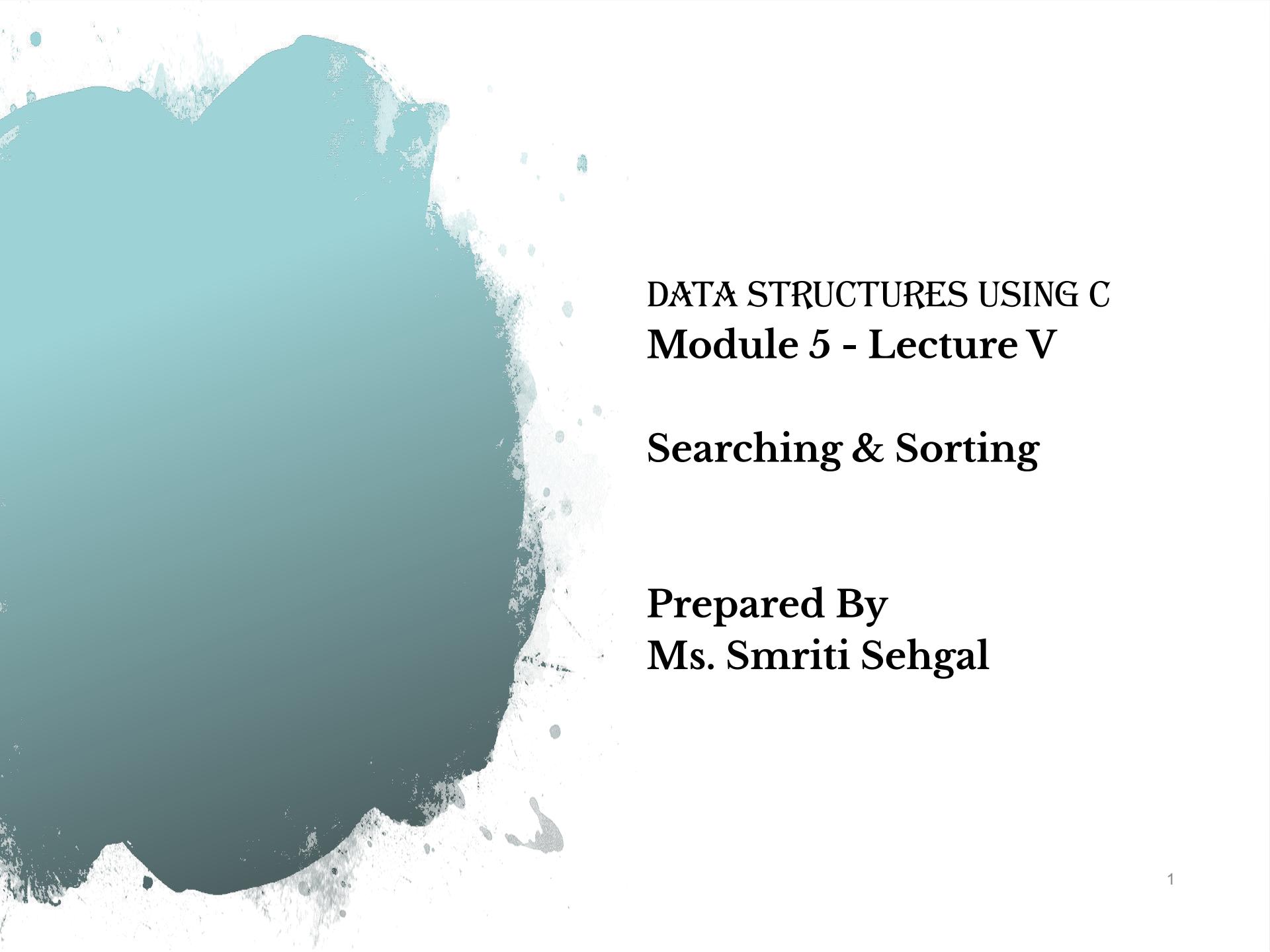
Comparing N and $\log_2 N$ performances

Array Size	Linear – N	Binary – $\log_2 N$
10	10	4
50	50	6
100	100	7
500	500	9
1000	1000	10
2000	2000	11
3000	3000	12
4000	4000	12
5000	5000	13
6000	6000	13
7000	7000	13
8000	8000	13
9000	9000	14
10000	10000	14

Important differences

- Input data needs to be sorted in Binary Search and not in Linear Search
- Linear search does the sequential access whereas Binary search access data randomly.
- Time complexity of linear search - $O(n)$, Binary search has time complexity $O(\log n)$.
- Linear search performs equality comparisons and Binary search performs ordering comparisons

Thank you



DATA STRUCTURES USING C

Module 5 - Lecture V

Searching & Sorting

Prepared By
Ms. Smriti Sehgal

Syllabus

- Insertion Sort, Bubble sort, Selection sort, Quick sort, Merge sort, Heap sort, Partition exchange sort, Shell sort, Sorting on different keys, External sorting. Linear search, Binary search, **Hashing: Hash Functions, Collision Resolution Techniques.**

In this session

We will talk about

- Hashing
- Hash Functions
- Collision Resolution Techniques

Searching

- Linear Search – $O(n)$
- Binary Search – $O(\log n)$

What if we want search operation to be $O(1)$? Solution: Hashing

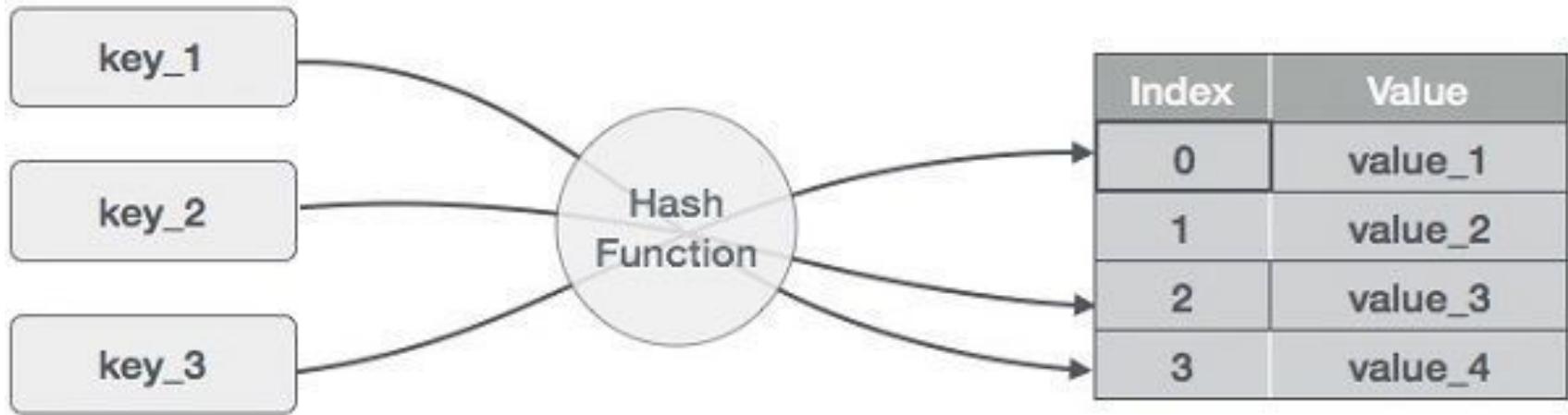
Hash Table is a data structure which stores data in an associative manner.

In this, data is stored in an array format, where each data value has its own unique index value.

Access of data becomes very fast if we know the index of the desired data.

Hash Table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from.

Hashing



Hashing is a technique to convert a range of key values into a range of indexes of an array.

Different Hash Functions

- Division Method
 - $H(x) = x \bmod M$
- Multiplication Method
 - $H(x) = m (kA \bmod 1)$ where $0 < A < 1$
- Mid-Square Method
 - $H(x) = s$ where $s =$ middle r digits of k^2

Consider, Modulo as Hash Function and Hash Table has size 20

Sr.No.	Key	Hash	Array Index
1	1	$1 \% 20 = 1$	1
2	2	$2 \% 20 = 2$	2
3	42	$42 \% 20 = 2$	2
4	4	$4 \% 20 = 4$	4
5	12	$12 \% 20 = 12$	12
6	14	$14 \% 20 = 14$	14
7	17	$17 \% 20 = 17$	17
8	13	$13 \% 20 = 13$	13
9	37	$37 \% 20 = 17$	17

- Since a hash function gets us a small number for a key which is a big integer or string, there is a possibility that two keys result in the same value.
- The situation where a newly inserted key maps to an already occupied slot in the hash table is called collision and must be handled using some collision handling technique.

- Open Addressing
- Chaining

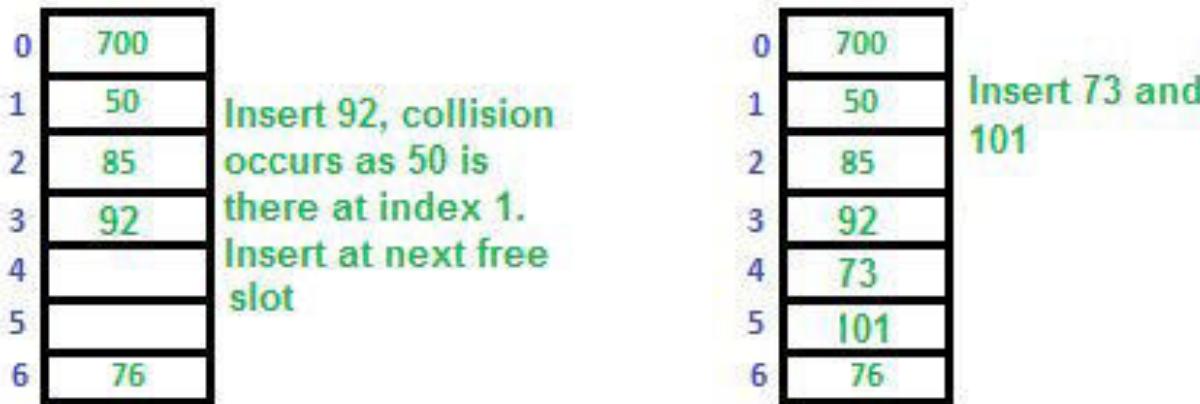
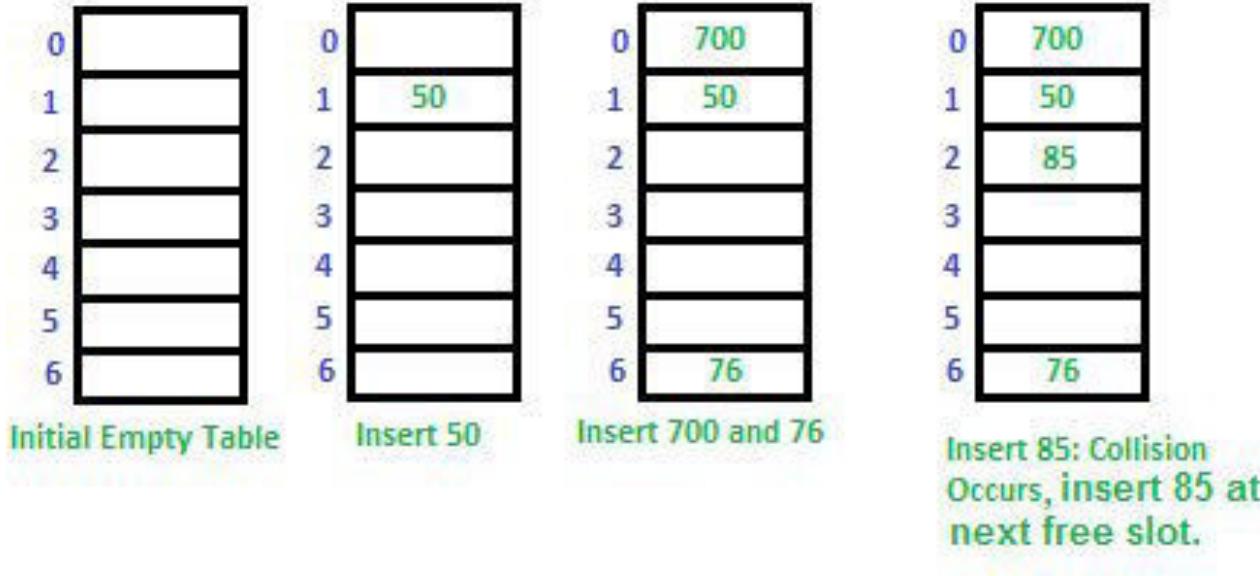
Collision Resolution by Addressing

- In Open Addressing, all elements are stored in the hash table itself. So at any point, size of the table must be greater than or equal to the total number of keys

- It may happen that the hashing technique creates an already used index of the array.
- In such a case, we can search the next empty location in the array by looking into the next cell until we find an empty cell.
- This technique is called linear probing.

Sr.No.	Key	Hash	Array Index	After Linear Probing, Array Index
1	1	$1 \% 20 = 1$	1	1
2	2	$2 \% 20 = 2$	2	2
3	42	$42 \% 20 = 2$	2	3
4	4	$4 \% 20 = 4$	4	4
5	12	$12 \% 20 = 12$	12	12
6	14	$14 \% 20 = 14$	14	14
7	17	$17 \% 20 = 17$	17	17
8	13	$13 \% 20 = 13$	13	13
9	37	$37 \% 20 = 17$	17	18

- Insert(k): Keep probing until an empty slot is found. Once an empty slot is found, insert k .
- Search(k): Keep probing until slot's key doesn't become equal to k or an empty slot is reached.
- Delete(k): ***Delete operation is interesting.*** If we simply delete a key, then search may fail. So slots of deleted keys are marked specially as "deleted". Insert can insert an item in a deleted slot, but the search doesn't stop at a deleted slot.



- **Quadratic Probing**, We look for i^2 th slot in i 'th iteration.
- let $\text{hash}(x)$ be the slot index computed using hash function.
- If slot $\text{hash}(x) \% S$ is full,
 - then we try $(\text{hash}(x) + 1*1) \% S$
 - If $(\text{hash}(x) + 1*1) \% S$ is also full,
 - then we try $(\text{hash}(x) + 2*2) \% S$
 - If $(\text{hash}(x) + 2*2) \% S$ is also full,
 - then we try $(\text{hash}(x) + 3*3) \% S$

Quadratic Probing Example

0	
1	
2	
3	
4	
5	
6	

Initial Empty Table

0	
1	50
2	
3	
4	
5	
6	

Insert 50

0	700
1	50
2	
3	
4	
5	
6	76

Insert 700
and 76

0	700
1	50
2	85
3	
4	
5	
6	76

Insert 85:

Collision occurs.

Insert at $1 + 1^2$ position

0	700
1	50
2	85
3	
4	
5	92
6	76

Insert 92:

Collision occurs at 1.

Collision occurs at $1 + 1^2$ position

Insert at $1 + 2^2$ position.

0	700
1	50
2	85
3	73
4	101
5	92
6	76

Insert 73 and 101

Let us consider a simple hash function as “key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.

Double Hashing

Amity School of Engineering & Technology

We use another hash function $\text{hash2}(x)$ and look for $i * \text{hash2}(x)$ slot in i 'th rotation.

let $\text{hash}(x)$ be the slot index computed using hash function.

If slot $\text{hash}(x) \% S$ is full,

then we try $(\text{hash}(x) + 1 * \text{hash2}(x)) \% S$

If $(\text{hash}(x) + 1 * \text{hash2}(x)) \% S$ is also full,

then we try $(\text{hash}(x) + 2 * \text{hash2}(x)) \% S$

If $(\text{hash}(x) + 2 * \text{hash2}(x)) \% S$ is also full,

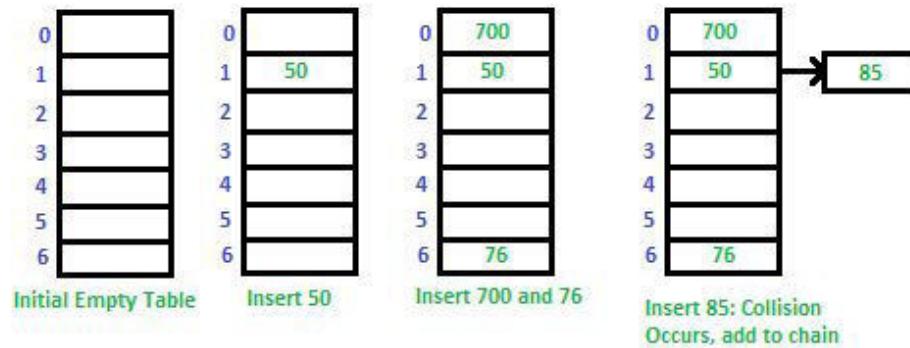
then we try $(\text{hash}(x) + 3 * \text{hash2}(x)) \% S$

Practice question

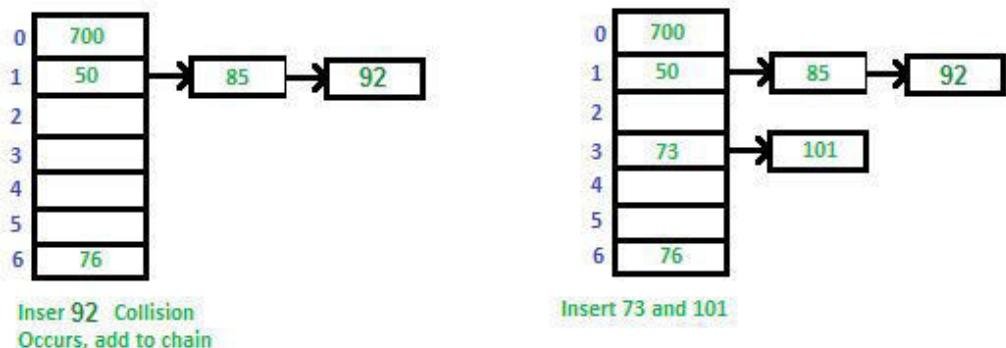
- Consider a hash table of size, 10. Using double hashing, insert the keys 72, 27, 36, 24, 63, 81, 92, 101 into the table.
Take $h_1 = (k \bmod 10)$ and $h_2 = (k \bmod 8)$

Collision Resolution by Chaining

The idea is to make each cell of hash table point to a linked list of records that have same hash function value.



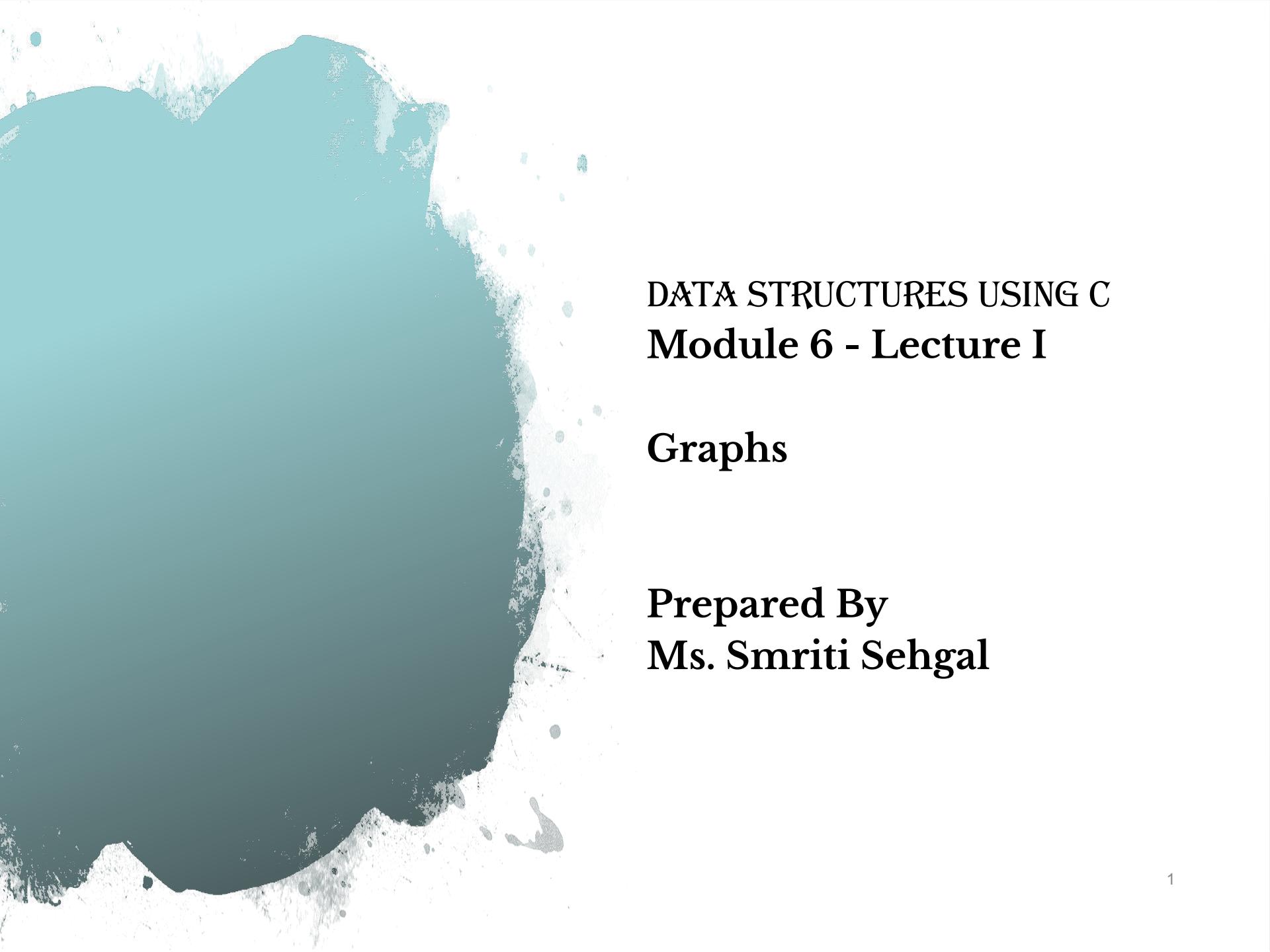
Let us consider a simple hash function as “**key mod 7**” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



Practice Question

- Insert the keys 7, 24, 18, 52, 36, 54, 11, 23 in aa chained hash table of 9 memory locations. Use $h(k) = k \bmod m$. ($m=9$)





DATA STRUCTURES USING C

Module 6 - Lecture I

Graphs

Prepared By
Ms. Smriti Sehgal

Graph Terminology

Graph

Graph is a non linear data structure,
set of points known as nodes (or vertices)
set of links known as edges (or Arcs)

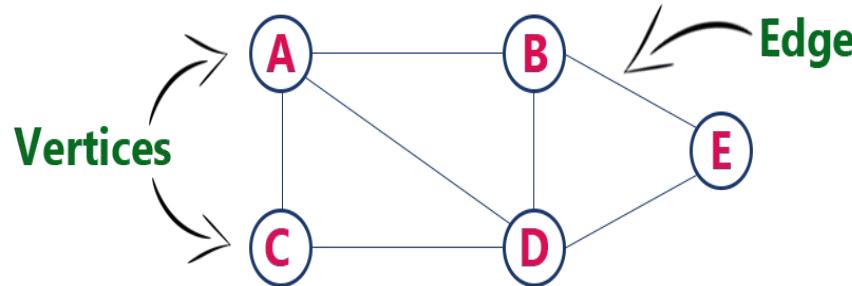
Graph is a collection of nodes and edges which connects nodes in the graph

Generally, a graph **G** is represented as

$$G = (V, E)$$

where **V** is set of vertices and **E** is set of edges.

Example :



The following is a graph with 5 vertices and 7 edges.

This graph G can be defined as $G = (V, E)$

Where $V = \{A, B, C, D, E\}$ and

$$E = \{(A, B), (A, C), (A, D), (B, D), (C, D), (B, E), (E, D)\}.$$

Graph Terminology

1. Vertex : A individual data element of a graph is called as Vertex. Vertex is also known as node. In above example graph, A, B, C, D & E are known as vertices.

2. Edge: An edge is a connecting link between two vertices. Edge is also known as Arc. An edge is represented as (startingVertex, endingVertex).

Edges are of three types.

Undirected Edge : (A,B) is equal to edge (B,A). **Bidirectional**

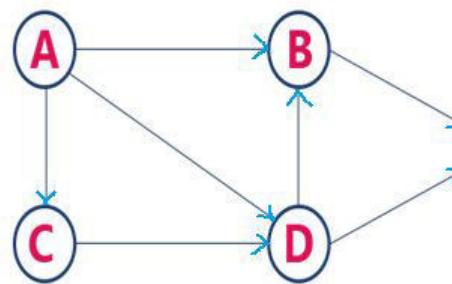
Directed Edge : (A,B) is not equal to edge (B,A). **Unidirectional**

Weighted Edge - An edge with cost on it.

Graph Terminology cont.

3. Undirected Graph :

A graph with only undirected edges



4. Directed Graph :

edges

5. Mixed Graph: A graph containing both directed and undirected edges

directed edges

6. Source and Destination Vertices : A vertex is said to be the source of an edge if it is its origin and the other vertex is said to be the destination of the edge.

Source, its first endpoint is the destination vertex.

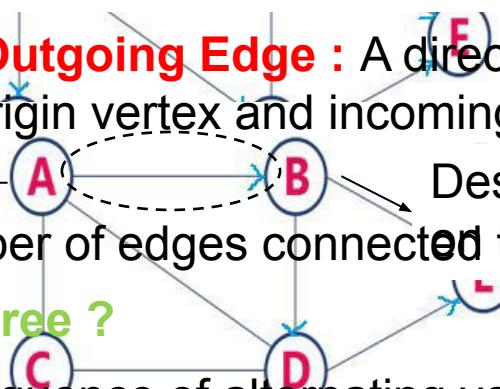
7. Incoming Edge & Outgoing Edge : A directed edge is said to be an outgoing edge on its origin vertex and incoming edge on its destination vertex.

Sour

Destinati

8. Degree : Total number of edges connected to a vertex

In-degree & Out-degree ?



9. Path : A path is a sequence of alternating vertices and edges

Graph Representations

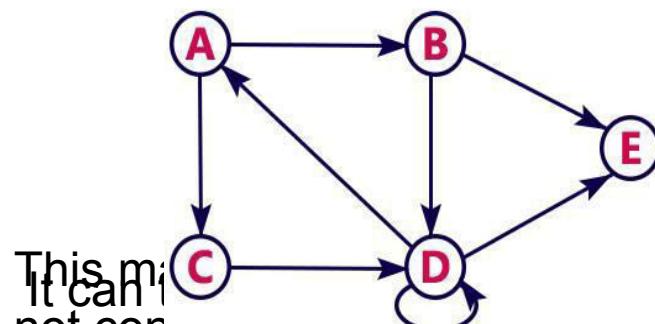
Adjacency Matrix

In this representation, graph can be represented using a matrix of size $V \times V$

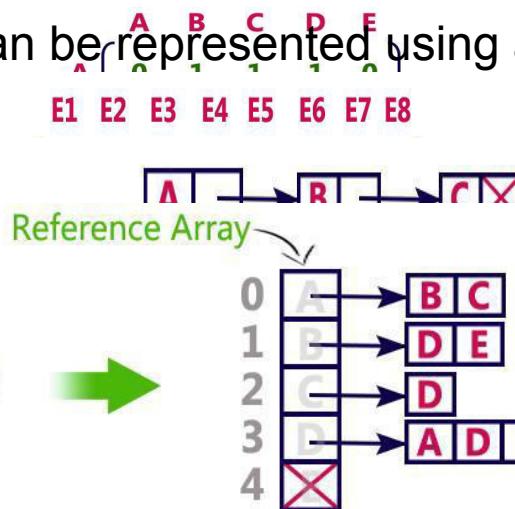
In this representation, graph can be represented using a matrix of size $V \times E$.

1. Using Linked List

2. Using array



No matter how few edges the graph has, the matrix takes $O(n^2)$ in memory



PATH Matrix / Reachability Matrix

Graph

If G is the simple graph

Terminology

- Different Operations for Graphs

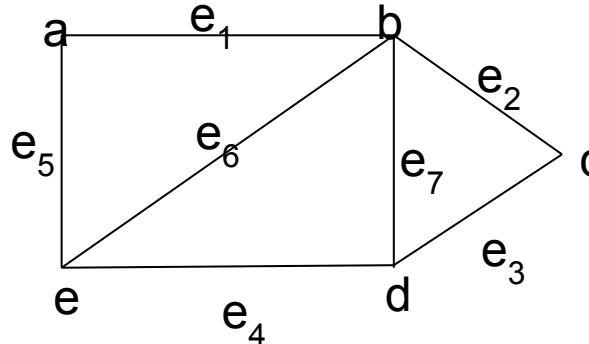
- Graph Traversal

- Minimum

spanning tree - Dijkstra shortest Path Algorithm

$$P_{ij} = \begin{cases} 1, & \text{if } j\text{th edge lies on } i\text{th path} \\ 0, & \text{otherwise} \end{cases}$$

Example



Find path matrix between vertex a & b , i.e. $P(a, d)$

We consider all possible path between a & d

$$(1). \{e_1, e_7\}$$

$$(2). \{e_1, e_6, e_4\}$$

$$(3). \{e_1, e_2, e_3\}$$

$$(4). \{e_5, e_6, e_7\}$$

$$(5). \{e_5, e_4\}$$

$$P(a, d) =$$

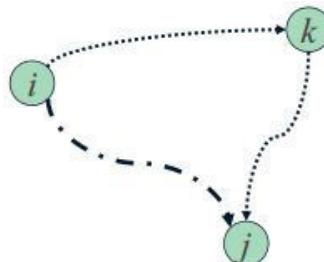
e_1	e_2	e_3	e_4	e_5	e_6	e_7
1	0	0	0	0	0	1
1	0	0	1	0	1	0
1	1	1	0	0	0	0
0	0	0	0	1	1	1
0	0	0	1	1	0	0

Warshall's Algorithm

Main idea: Main idea: a path exists between two vertices i, j , iff

- there is an edge from i to j ; or
- there is a path from i to j going through vertex 1; or
- there is a path from i to j going through vertex 1 and/or 2; or
- ...
- there is a path from i to j going through any of the other vertices

On the k^{th} iteration, the algorithm determine if a path exists between two vertices i, j using just vertices among $1, \dots, k$ allowed as intermediate



$$R^{(k)}[i,j] = \begin{cases} R^{(k-1)}[i,j] & \text{(path using just } 1, \dots, k-1\text{)} \\ \text{or} \\ (R^{(k-1)}[i,k] \text{ and } R^{(k-1)}[k,j]) & \text{(path from } i \text{ to } k \text{ and from } k \text{ to } j \text{ using just } 1, \dots, k-1\text{)} \end{cases}$$

Pseudo code for Warshall's Algorithm

1. Create a Adjacency Matrix

```
Repeat for I, j = 1, 2, ..., n    // initialize p
    if (A[i,j] = 0 then p[i, j]=0)
    else p[i, j]=1               // End of loop
                                // Update p
```

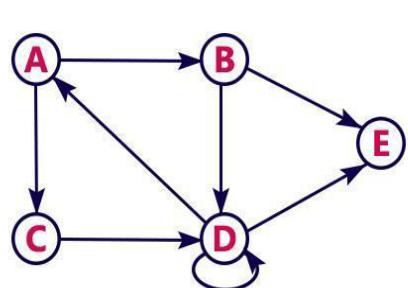
2. Repeat for k = 1, 2, ..., n

```
    Repeat for i = 1, 2, ..., n
        Repeat for j = 1, 2, ..., n
            p[i, j] = p[i, j] or (p[ i,k] and p[k, j])
        Exit
```

Time efficiency: $\theta(n^3)$

Space efficiency: Matrices can be written over their predecessors

Warshall's Algorithm cont'd



Find the path matrix for the above graph

$p[i, j] = p[i, j] \text{ or } (p[i, k] \text{ and } p[k, j])$

Or

$R^{(0)}[i, j]$	A/1	B/2	C/3	D/4	E/5
$R^{(1)}[A/1]$	0	1	1	0	0
$R^{(1)}[B/2]$	0	0	0	1	1
$R^{(1)}[C/3]$	0	0	0	1	0
$R^{(1)}[D/4]$	1	0	0	1	1
$R^{(1)}[E/5]$	0	0	0	0	0

$$R^{(1)}[i, j] = R^{(0)}[i, j] \text{ or } (R^{(0)}[i, k] \text{ and } R^{(0)}[k, j])$$

$$R^{(1)} =$$

	1	2	3	4	5
1	0	1	1	0	0
2	0	0	0	1	1
3	0	0	0	1	0
4	1	1	1	1	1
5	0	0	0	0	0

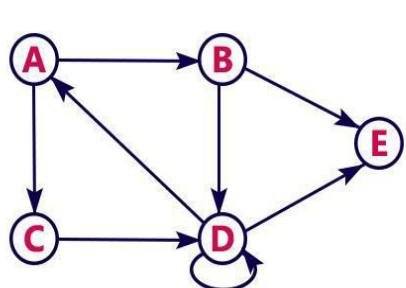
$$R^{(2)} =$$

	1	2	3	4	5
1	0	1	1	1	1
2	0	0	0	1	1
3	0	0	0	1	0
4	1	1	1	1	1
5	0	0	0	0	0

$$R^{(3)} =$$

	1	2	3	4	5
1	0	1	1	1	0
2	0	0	0	1	1
3	0	0	0	1	0
4	1	1	1	1	1
5	0	0	0	0	0

Warshall's Algorithm cont'd



Find the path matrix for the above graph

$p[i, j] = p[i, j] \text{ or } (p[i, k] \text{ and } p[k, j])$

$$R^{(3)}[i, j] = R^{(0)}[i, j] \text{ or } (R^{(0)}[i, k] \text{ and } R^{(0)}[k, j])$$

	1	2	3	4	5
1	0	1	1	1	0
2	0	0	0	1	0
3	0	0	0	1	0
4	1	1	1	1	1
5	0	0	0	0	0

$$R^{(4)} =$$

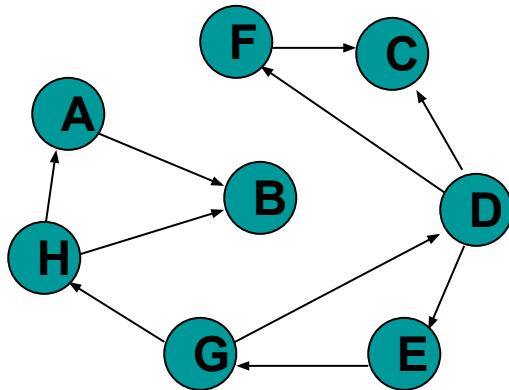
	1	2	3	4	5
1	1	1	1	1	1
2	1	0	1	1	1
3	1	0	1	1	1
4	1	1	1	1	1
5	0	0	0	0	0

$$R^{(5)} =$$

	1	2	3	4	5
1	1	1	1	1	1
2	1	0	1	1	1
3	1	0	1	1	0
4	1	1	1	1	1
5	0	0	0	0	0

This is the final path matrix

Exercise



Find the path matrix for the above graph

Topological Sorting

an ordering of the vertices in a directed acyclic graph, such that: If there is a path from u to v, then v appears after u in the ordering.

Conditions for Topological ordering

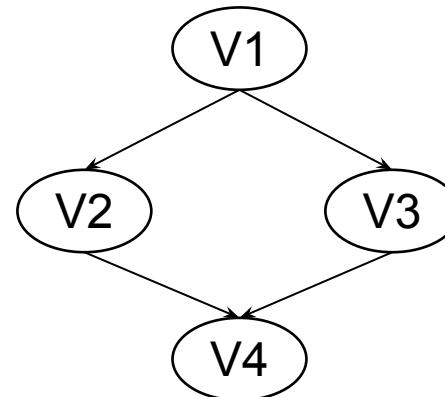
The graphs should be **directed**

The graphs should be **acyclic**

Example

:

V1 V2 V3 V4 & V1 V3 V2 V4



How to sort

The algorithm for topological sort uses "indegrees" of vertices.

Indegree of vertex: no.of incoming edges

Algorithm

1. Compute the indegrees of all vertices
2. Find a vertex **U** with indegree 0 and print it (store it in the ordering) If there is no such vertex then there is a cycle and the vertices cannot be ordered. Stop.
3. Remove **U** and all its edges **(U,V)** from the graph.
4. Update the indegrees of the remaining vertices.
5. Repeat steps 2 through 4 while there are vertices to be processed.

Example

1. Compute the indegrees:

V1: 0

V2: 1

V3: 2

V4: 2

V5: 2

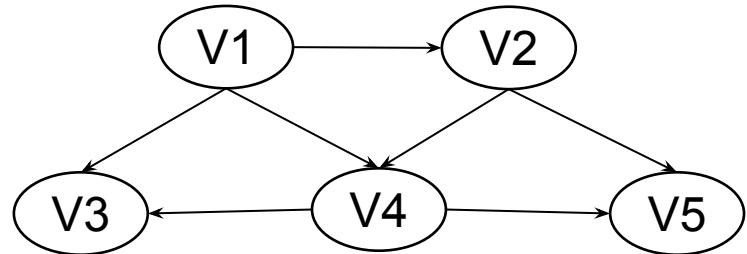
2. Find a vertex with indegree V1

0:

3. Output V1 , remove V1, so all the edges from V1 will also be deleted,
Now again update the indegrees

4. Follow step 2&3 till all vertex is removed.

Note: both V3 & V5 now have same indegree, here you can choose randomly any one.



	Indegree					
Sorted		V 1	V1,V2	V1,V2,V4	V1,V2,V4,V 3	V1,V2,V4,V3,V 5
V1	0					
V2	1	0				
V3	2	1	1	0		
V4	2	1	0			
V5	2	2	1	0	0	

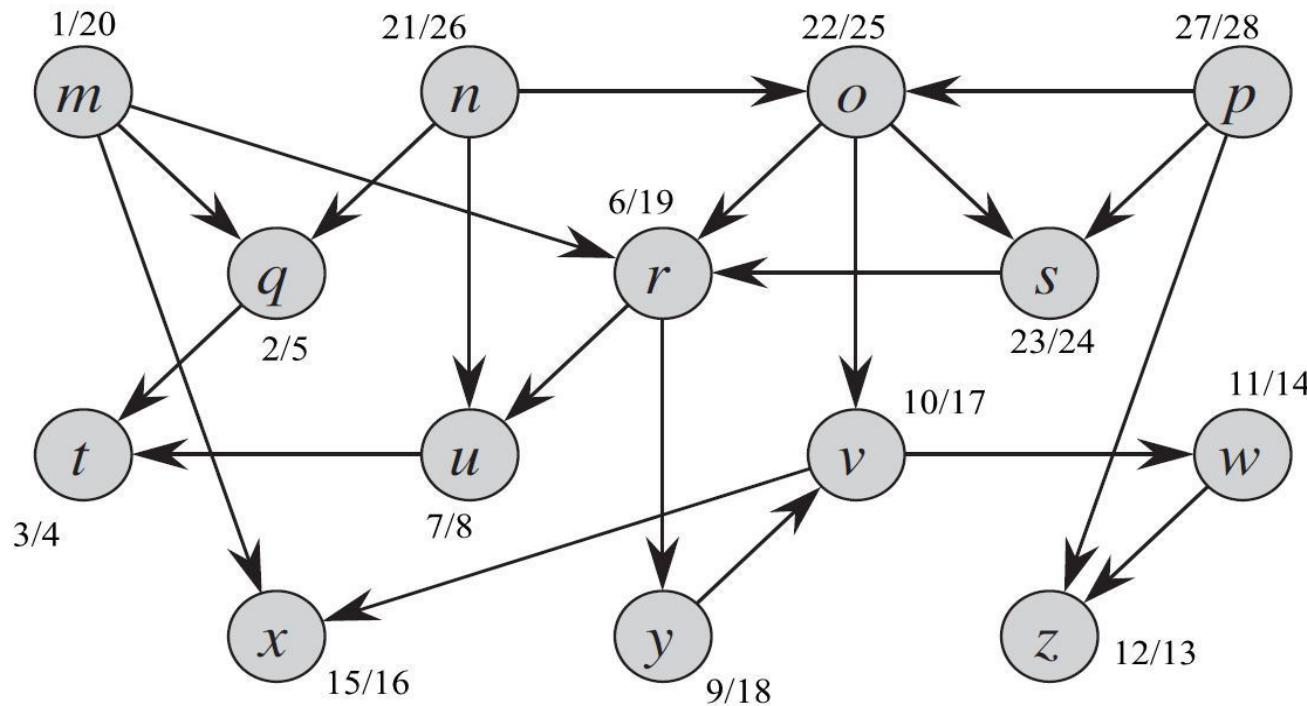
Complexity Analysis

Complexity of this algorithm: $O(|V|^2)$,

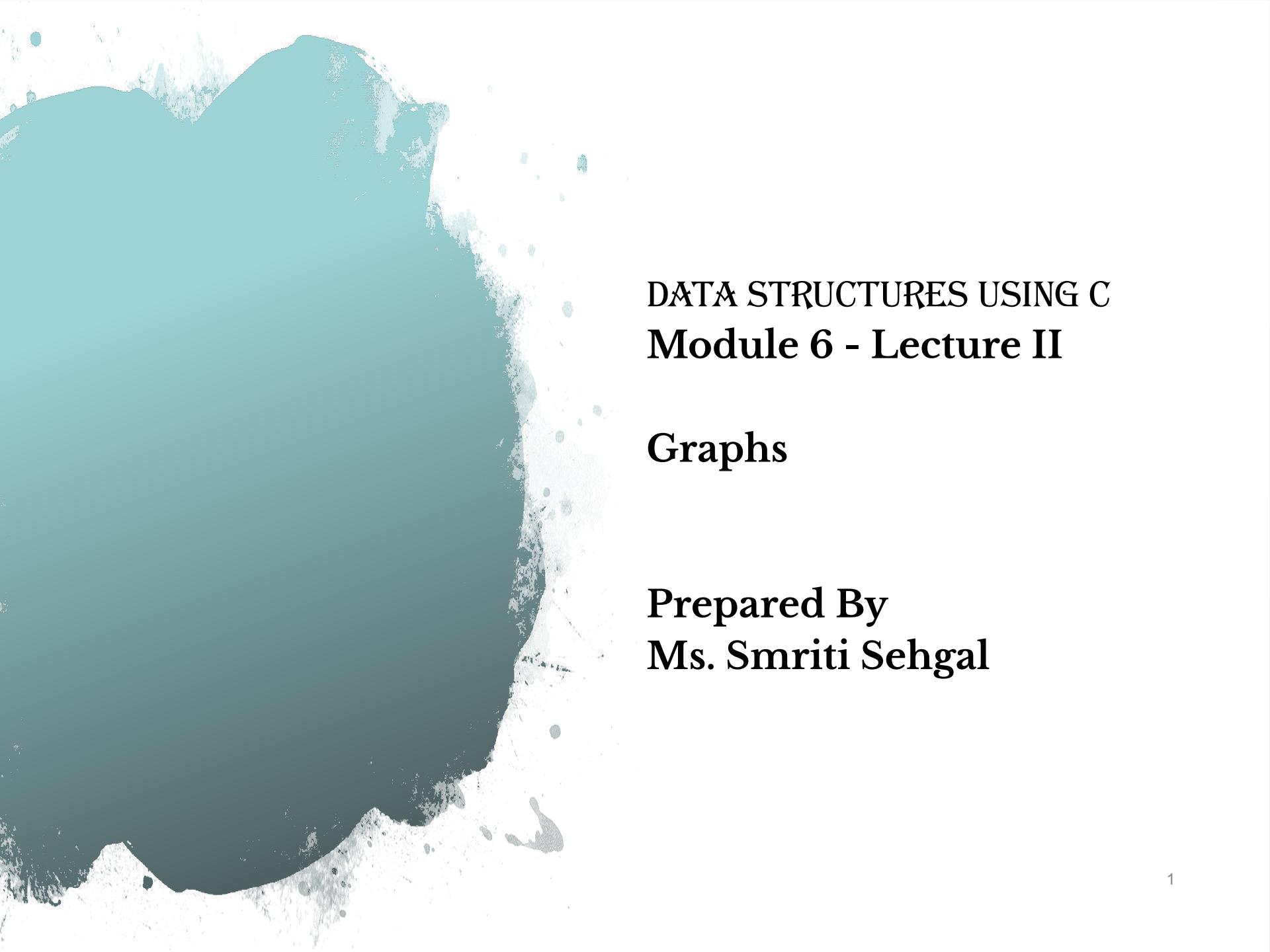
To find a vertex of indegree 0 we scan all the vertices- $|V|$ operations.

We do this for all vertices so $|V|^2$

Exercise







DATA STRUCTURES USING C

Module 6 - Lecture II

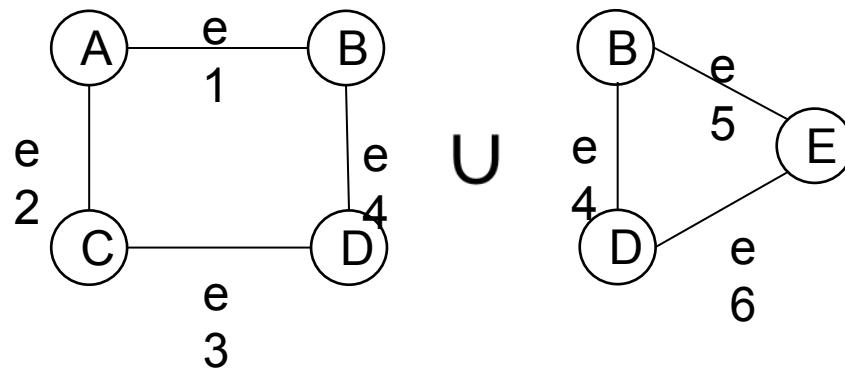
Graphs

Prepared By
Ms. Smriti Sehgal

Operations on Graph

1. Union If two graph G_1 & G_2 are given then $G_1 \cup G_2$

Will be $V(G_1 \cup G_2) = V(G_1) \cup V(G_2)$
& $E(G_1 \cup G_2) = E(G_1) \cup E(G_2)$

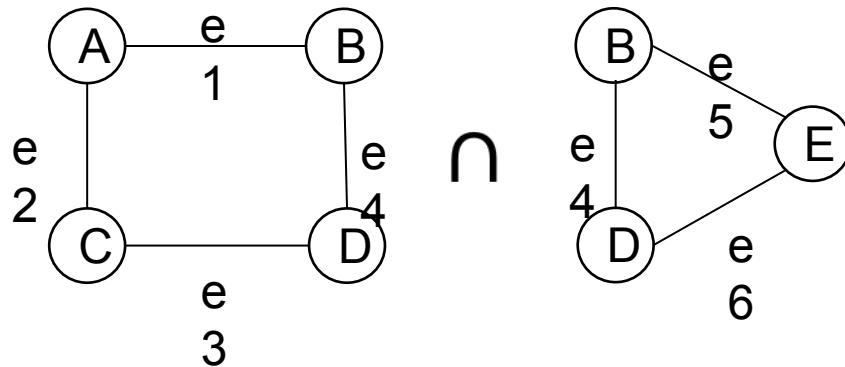


Operations on Graph

2. Intersection If two graph G_1 & G_2 are given then $G_1 \cap G_2$

Will be $V(G_1 \cap G_2) = V(G_1) \cap V(G_2)$

& $E(G_1 \cap G_2) = E(G_1) \cap E(G_2)$

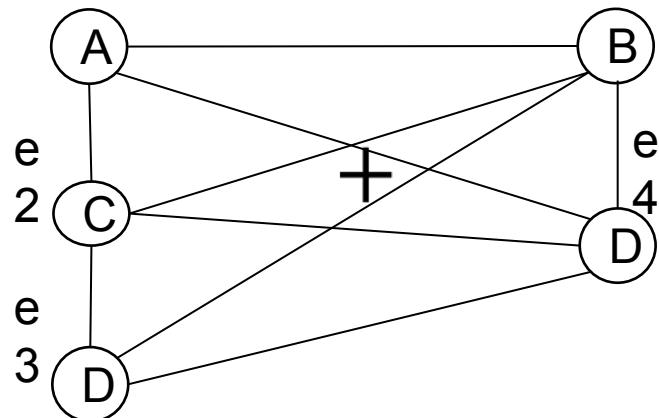


Operations on Graph

3. Sum of two graph If two graph G_1 & G_2 are given then $G_1 + G_2$

Will be $V(G_1 + G_2) = V(G_1) + V(G_2)$

& $E(G_1 + G_2)$ = Edges which are in G_1 & G_2 and edges obtained by joining each vertex of G_1 to each G_2



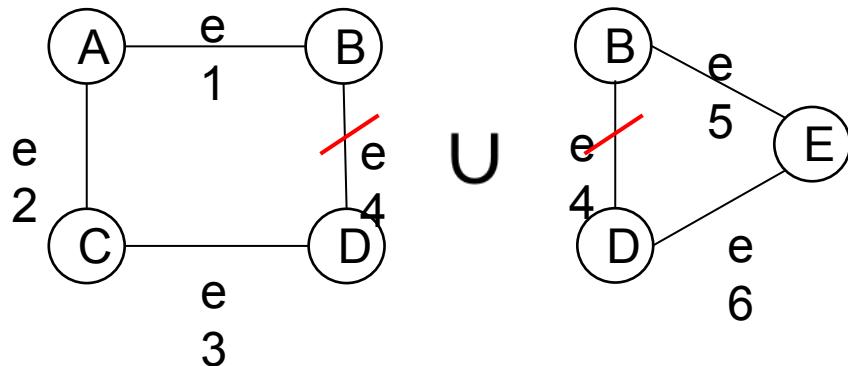
Operations on Graph

3. Ring Sum If two graph G_1 & G_2 are given then $G_1 \oplus G_2$

Will be $V(G_1 \oplus G_2) = V(G_1) \cup V(G_2)$

& $E(G_1 \oplus G_2) = E(G_1) \cup E(G_2) - E(G_1) \cap E(G_2)$

edges either in G_1 or G_2 but not in both

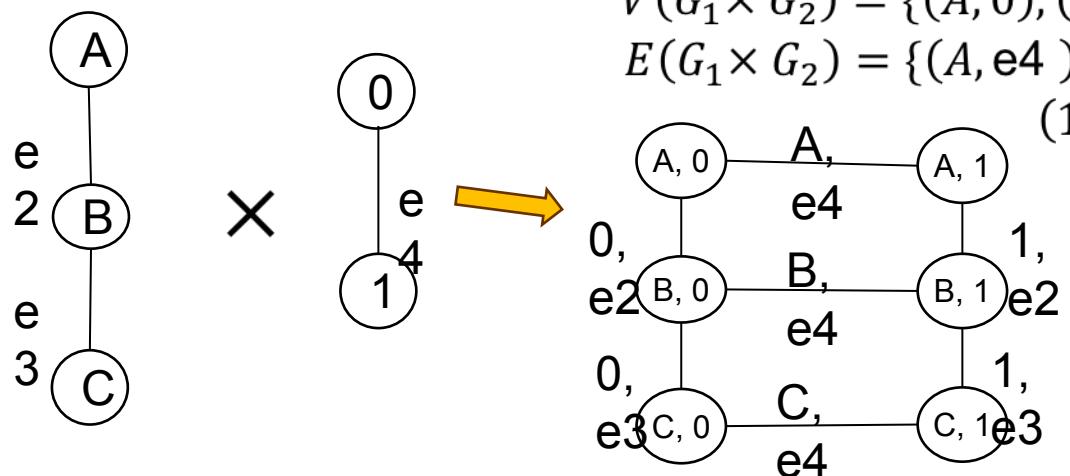


Operations on Graph

4. Product If two graph G_1 & G_2 are given then $G_1 \times G_2$

Will be $V(G_1 \times G_2) = V(G_1) \times V(G_2)$

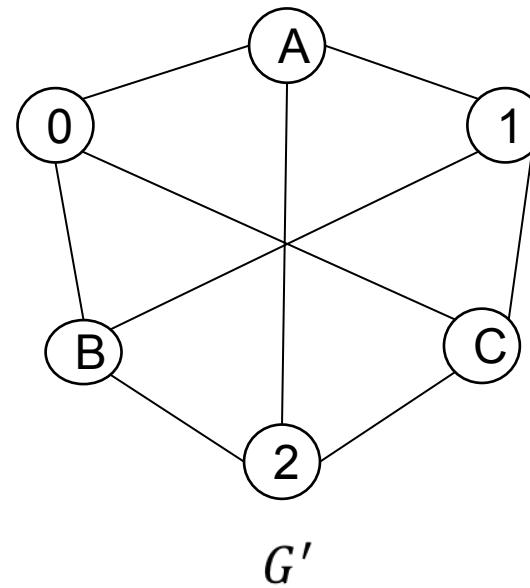
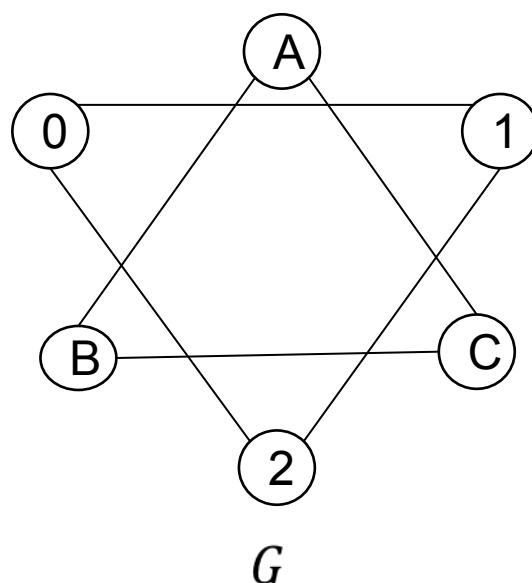
& $E(G_1 \times G_2) = E(G_1) \times E(G_2) \cup E(G_2) \times E(G_1)$



Operations on Graph

5. Complement $G \rightarrow G'$

G' is said to be complement when the vertex pair which are adjacent in G' are not adjacent in G .

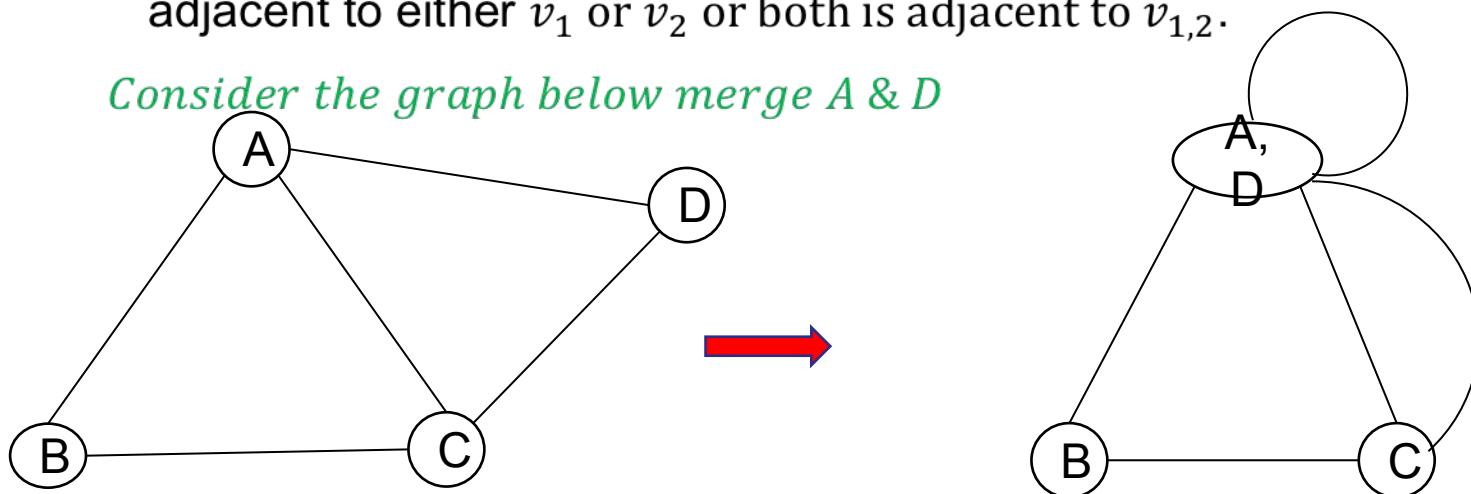


Operations on Graph

6. Fusion / Merged / Identified

v_1 & v_2 in graph G is said to be fused if these two vertices are replaced by a single new vertex $v_{1,2}$ such that every edge that is adjacent to either v_1 or v_2 or both is adjacent to $v_{1,2}$.

Consider the graph below merge A & D



Graph Traversal

Graph traversal is technique used for searching a vertex in a graph. The graph traversal is also used to decide the order of vertices to be visit in the search process.

DFS (Depth First Search)

BFS (Breadth First Search)

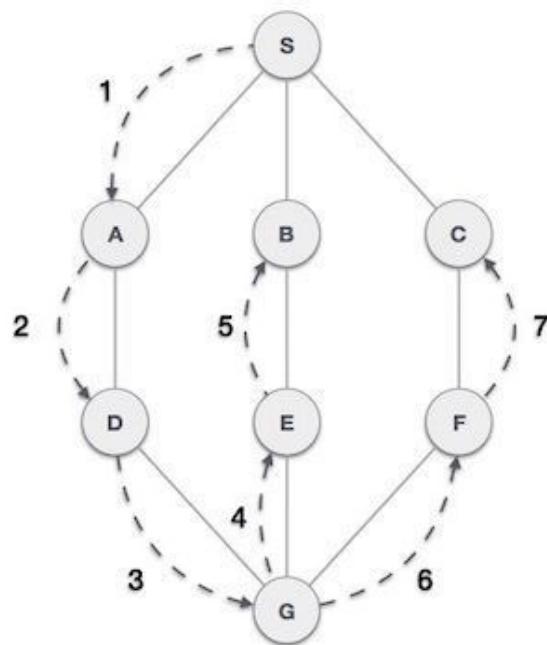
DFS (Depth First Search)

DFS traversal of a graph, **produces a spanning tree** as final result.

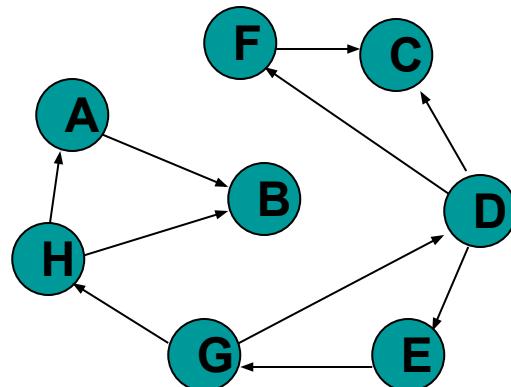
Spanning Tree is a graph **without any loops**.

We use **Stack data structure** with maximum size of total number of vertices in the graph.

Back tracking is coming back to the vertex from which we came to current vertex.



DFS (Depth First Search) Work-Through



The order nodes are visited:

DCEGHABF

Visited Array Stack

A	✓
B	✓
C	✓
D	✓
E	✓
F	✓
G	✓
H	✓

B
A
H
G
E
D

This diagram illustrates the Depth-First Search (DFS) algorithm's work-through process. It shows the visited array and the stack at each step of the search. The visited array indicates which nodes have been explored, while the stack shows the path taken by the algorithm. The order of nodes visited is DCEGHABF.

Algorithm for DFS

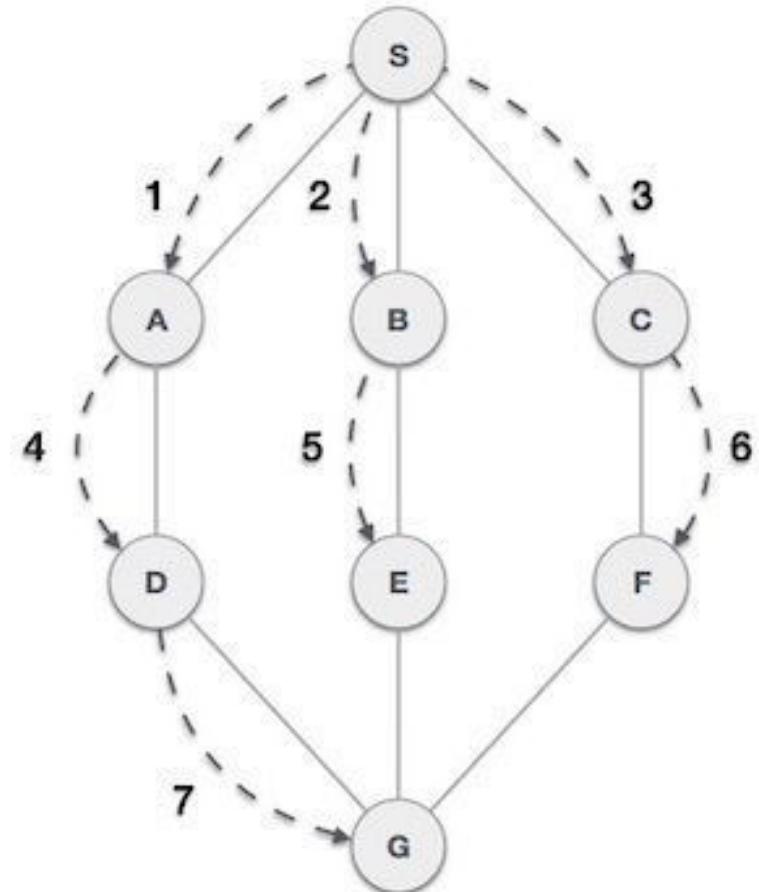
- **Step 1:** Define a Stack of size total number of vertices
- **Step 2:** Select any vertex as **starting point**. Visit that vertex and push it on to the Stack.
- **Step 3:** Visit any one of the **adjacent** vertex of the vertex which is at **top of the stack** which is **not visited and push** it on to the stack.
- **Step 4:** Repeat step 3 until there are no new vertex to be visit from the vertex on top of the stack.
- **Step 5:** When there is no new vertex to be visit then use **back tracking** and pop one vertex from the stack.
- **Step 6:** Repeat steps 3, 4 and 5 until stack becomes Empty. When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

Pseudo code for DFS

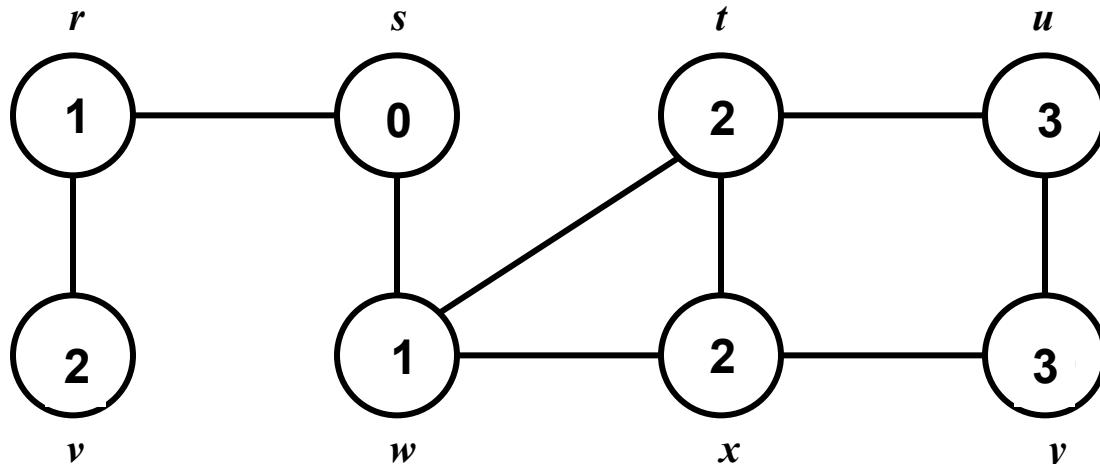
```
DFS(input: Graph G)
{
    Stack S; Integer x, y;
    while (G has an unvisited node x)
    {
        visit(x);
        push(x,S);
        while (S is not empty)
        {
            t := top(S);
            if (t has an unvisited neighbor y)
                visit(y);
                push(y,S);
            else
                pop(S);
        }
    }
}
```

Breadth First Search Traversal

- BFS traversal of a graph, produces a **spanning tree** as final result.
- **Spanning Tree** is a graph without any loops.
- We use **Queue data structure** with maximum size of total number of vertices in the graph



Example


Q[8]

s	w	r	t	x	v	u	y
---	---	---	---	---	---	---	---

1. Choose any random vertex as start point.
2. Insert that vertex in a Queue
3. Insert adjacent neighbour in a Queue and Deque
4. Next vertex in queue is selected again step 3 is carried out. Continue till all vertex been visited
5. We have visited all vertex and also queue is empty.
6. The path we followed is the **final Spanning tree**.

Algorithm for BFS

- Rule 1 – Visit adjacent unvisited vertex. Mark it visited. Display it. Insert it in a queue.
- Rule 2 – If no adjacent vertex found, remove the first vertex from queue.
- Rule 3 – Repeat Rule 1 and Rule 2 until queue is empty.

Pseudo code for DFS

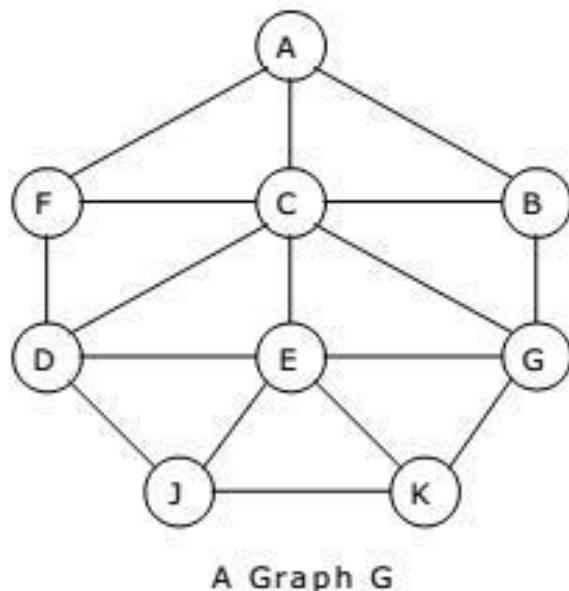
```
BFS(input: graph G)
{
    Queue Q; Integer x, z, y;
    while (G has an unvisited node x)
    {
        visit(x); Enqueue(x,Q);
        while (Q is not empty)
        {
            z := Dequeue(Q);
            for all (unvisited neighbor y of z)
            {
                visit(y);
                Enqueue(y,Q);
            }
        }
    }
}
```

Complexity Analysis

- Queuing time is $O(V)$ and scanning all edges requires $O(E)$
- Overhead for initialization is $O(V)$
- So, total running time is $O(V+E)$

Example 1:

Consider the graph shown below. Traverse the graph shown below in breadth first order and depth first order.



Node	Adjacency List
A	F, C, B
B	A, C, G
C	A, B, D, E, F, G
D	C, F, E, J
E	C, D, G, J, K
F	A, C, D
G	B, C, E, K
J	D, E, K
K	E, G, J

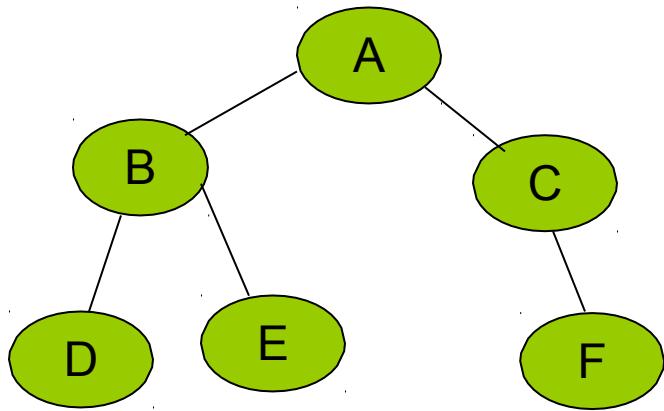
Adjacency list for graph G





Minimum Spanning Tree

TREE



- Connected acyclic graph
- Tree with n nodes contains exactly $n-1$ edges.

GRAPH



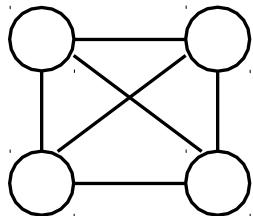
- Graph with n nodes contains less than or equal to $n(n-1)/2$ edges.

SPANNING TREE...

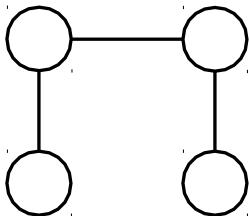
Suppose you have a connected undirected graph

- Connected: every node is reachable from every other node
- Undirected: edges do not have an associated direction

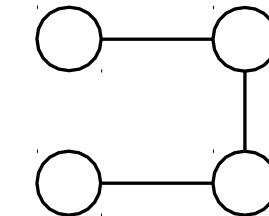
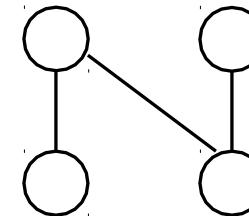
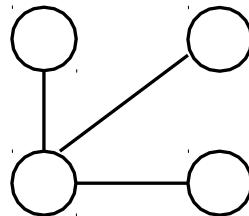
...then a **spanning tree** of the graph is a connected subgraph in which there are no cycles



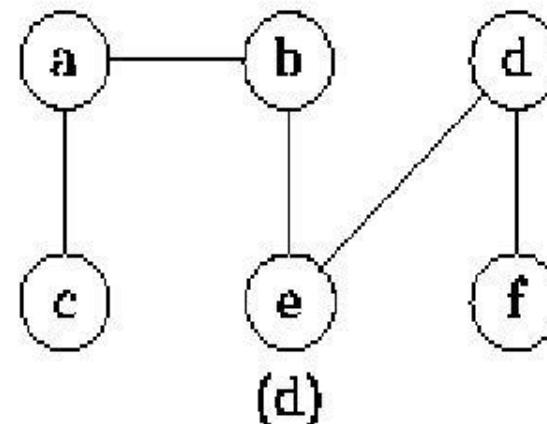
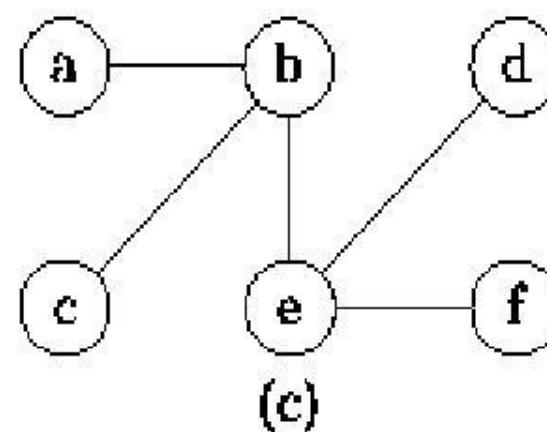
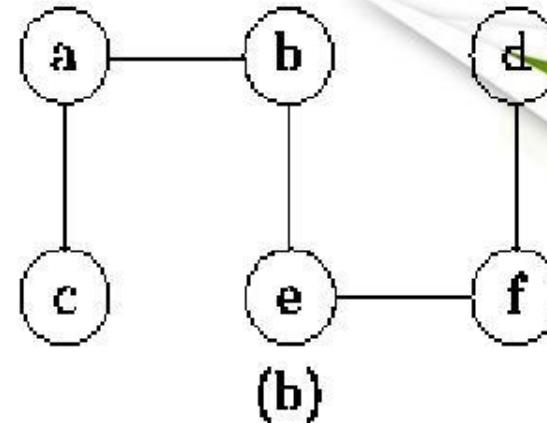
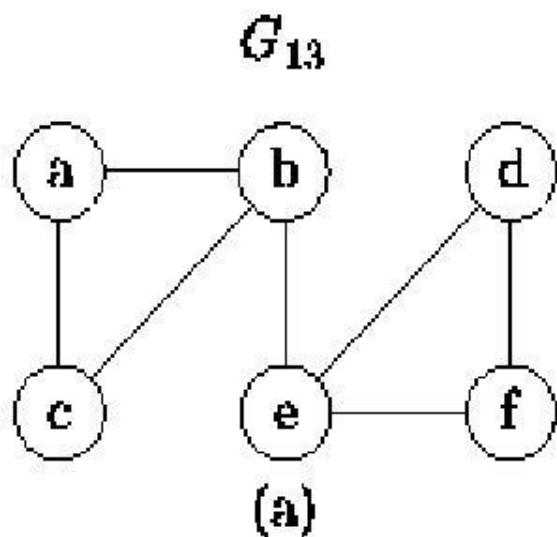
A connected,
undirected
graph



Four of the spanning trees of the graph



EXAMPLE..



Minimizing costs

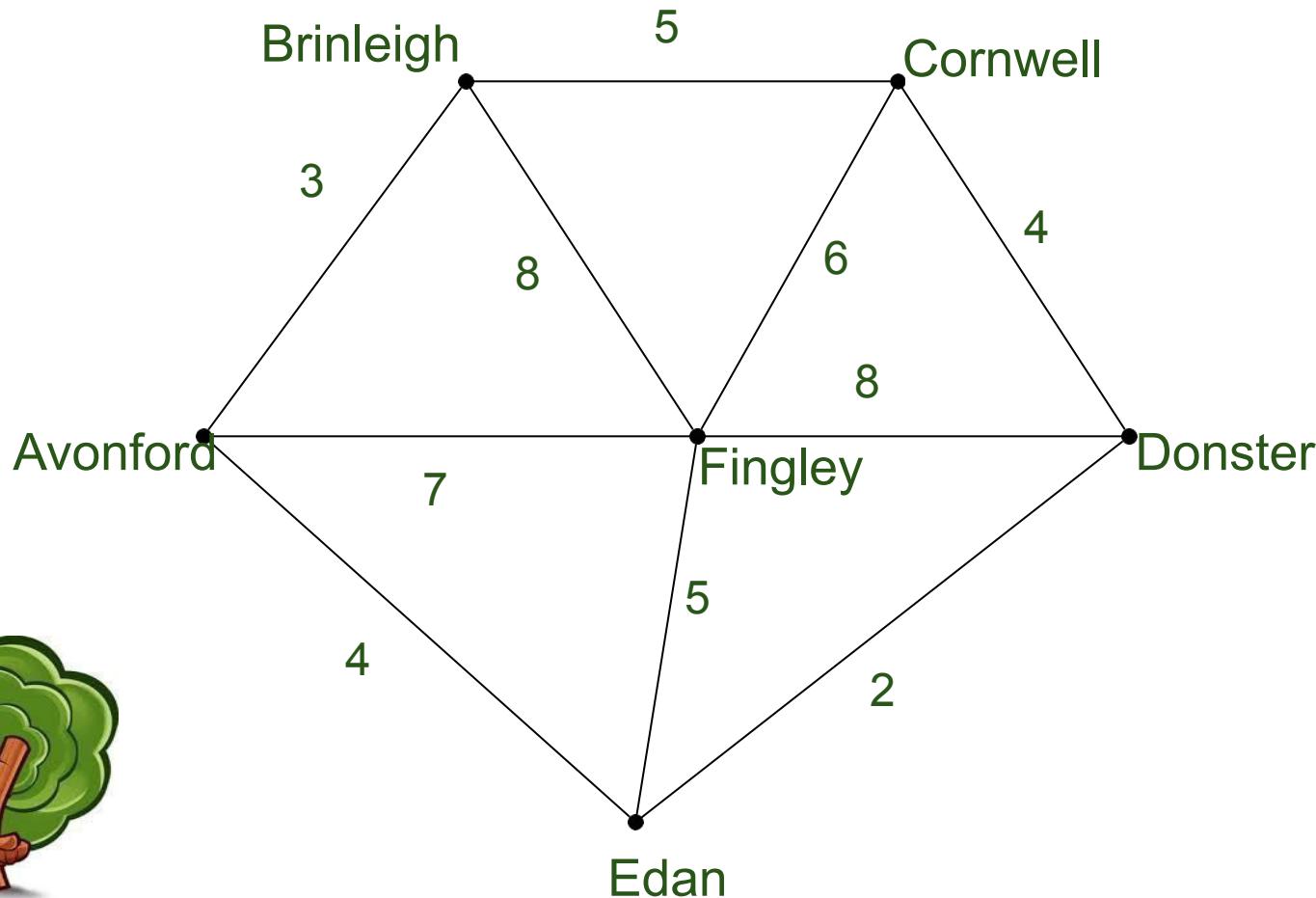
Suppose you want to supply a set of houses (say, in a new subdivision) with:

- electric power
- water
- sewage lines
- telephone lines
- To keep costs down, you could connect these houses with a spanning tree (of, for example, power lines)
- However, the houses are not all equal distances apart. To reduce costs even further, you could connect the houses with a minimum-cost spanning tree



Example

A cable company want to connect five villages to their network which currently extends to the market town of Avonford. What is the minimum length of cable needed?



MINIMUM SPANNING TREE

Let $G = (N, A)$ be a connected, undirected graph where N is the set of nodes and A is the set of edges. Each edge has a given nonnegative length. The problem is to find a subset T of the edges of G such that all the nodes remain connected when only the edges in T are used, and the sum of the lengths of the edges in T is as small as possible possible. Since G is connected, at least one solution must exist.

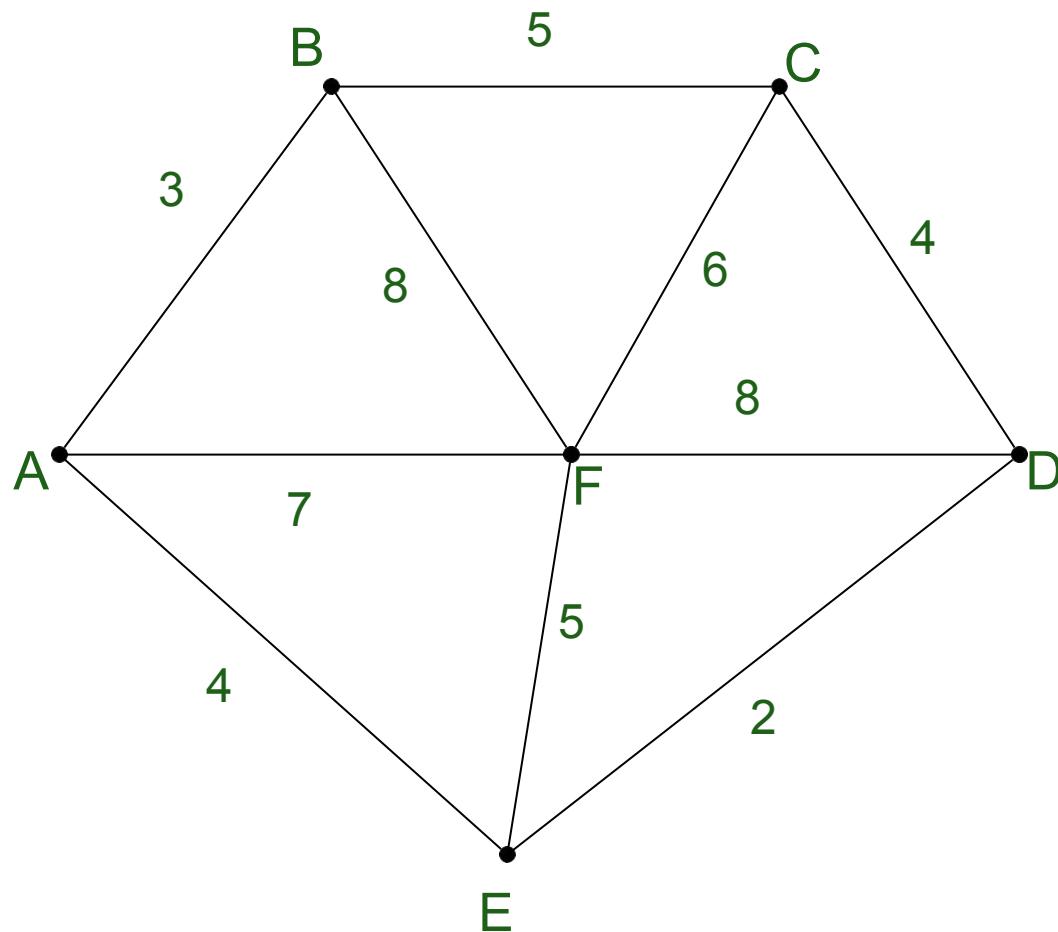


Finding Spanning Trees

- There are two basic algorithms for finding minimum-cost spanning trees, and both are greedy algorithms
- **Kruskal's algorithm:**
Created in 1957 by Joseph Kruskal
- **Prim's algorithm**
Created by Robert C. Prim

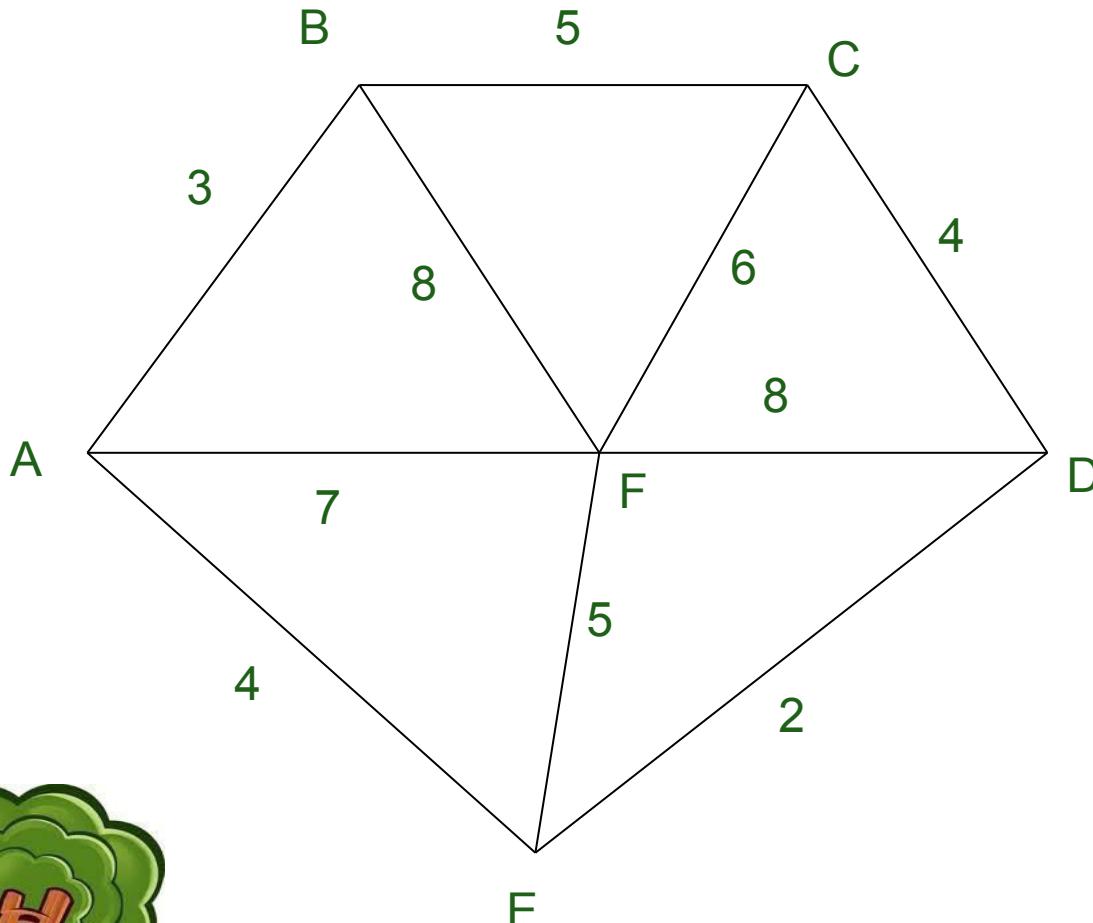


We model the situation as a network, then the problem is to find the minimum connector for the network

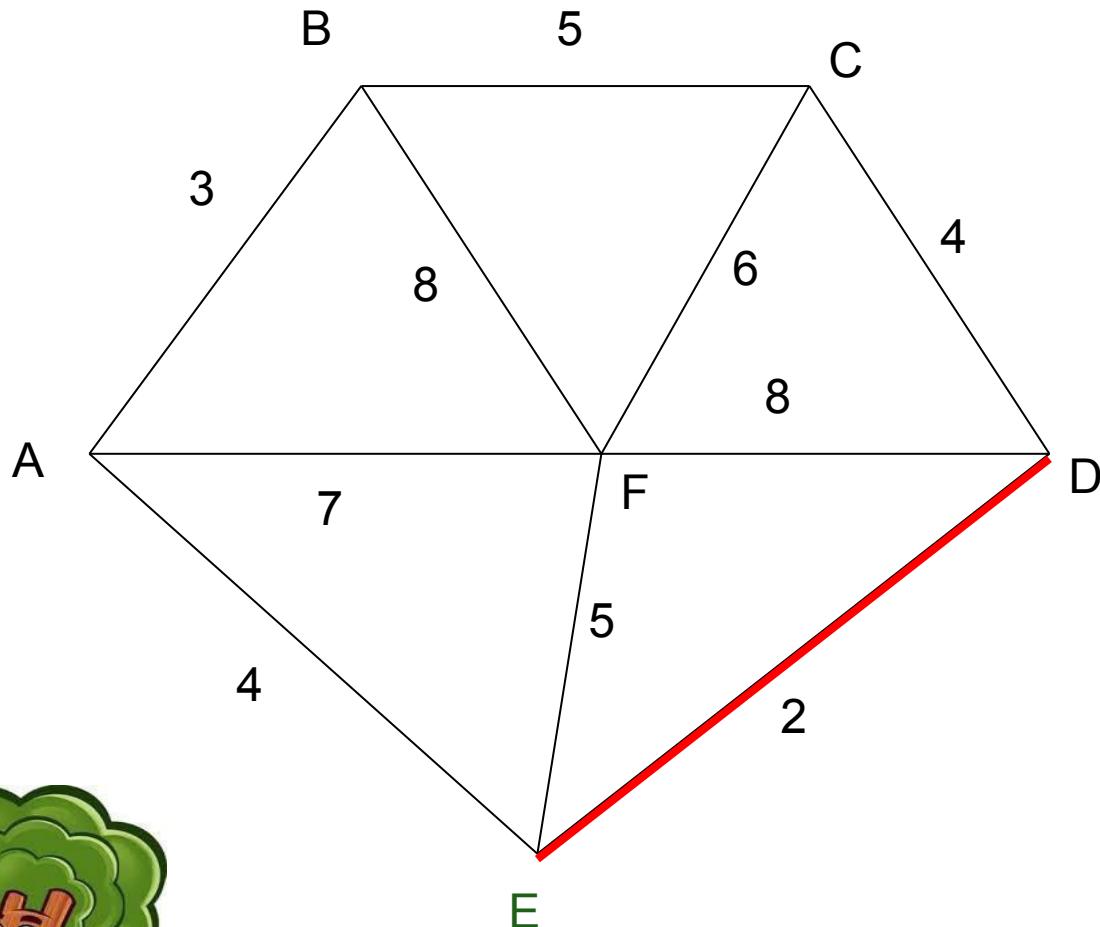


Kruskal's Algorithm

List the edges in order of size:



Kruskal's Algorithm

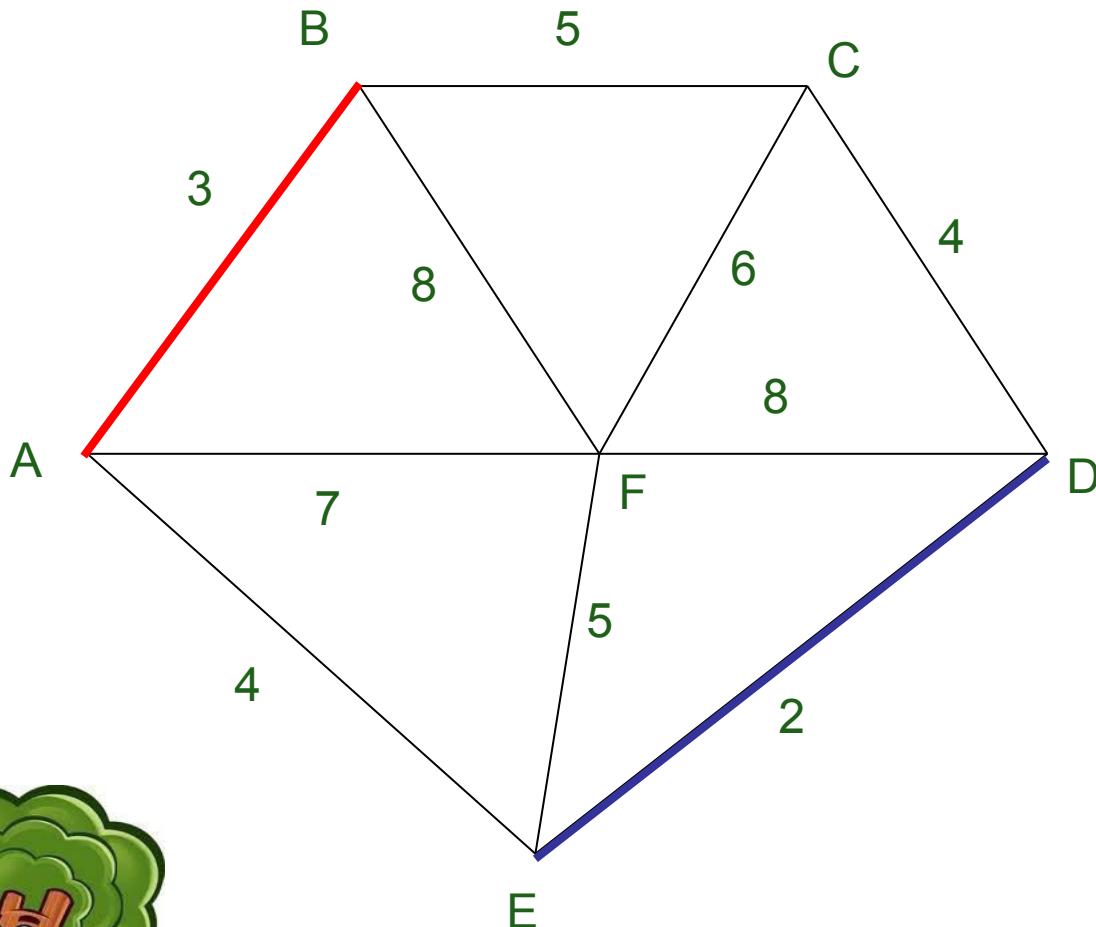


Select the shortest edge in the network

ED 2



Kruskal's Algorithm

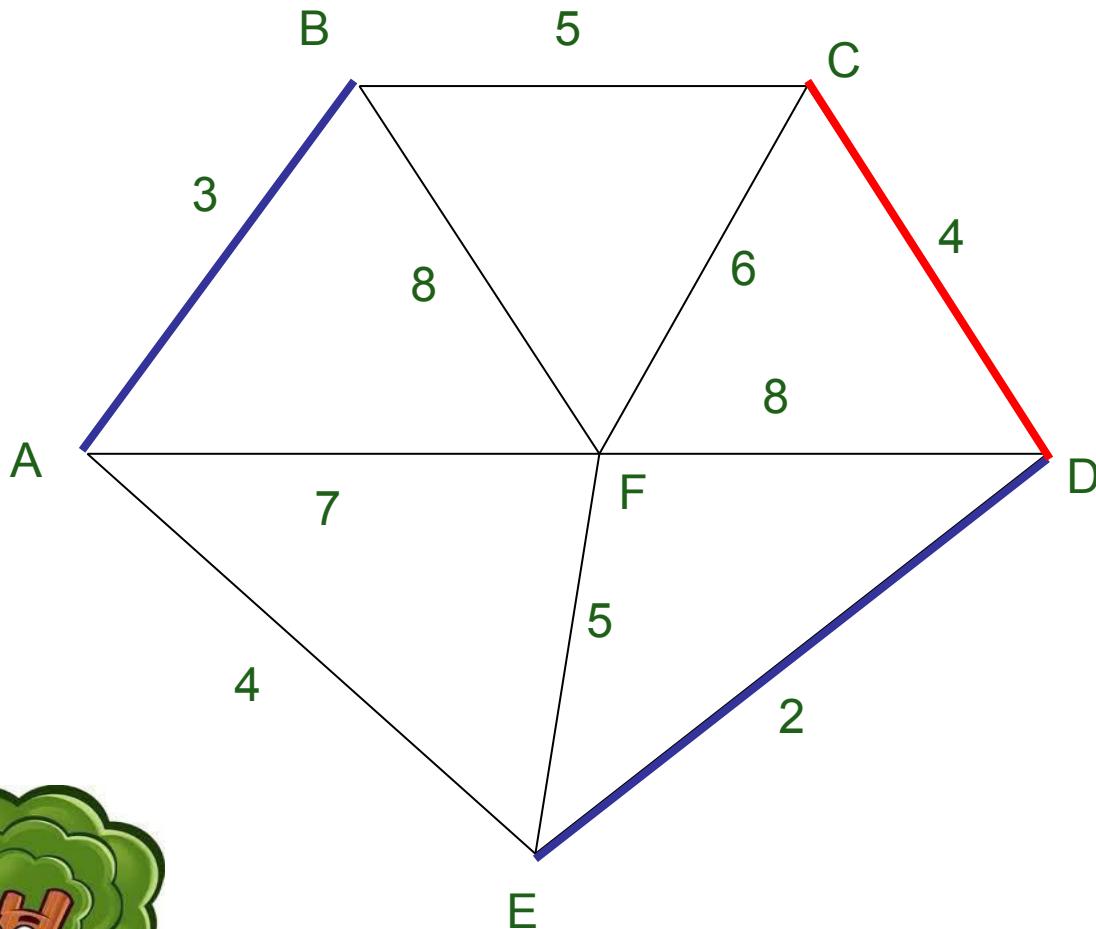


Select the next
shortest
edge which does not
create a cycle

ED 2
AB 3



Kruskal's Algorithm



Select the next shortest edge which does not create a cycle

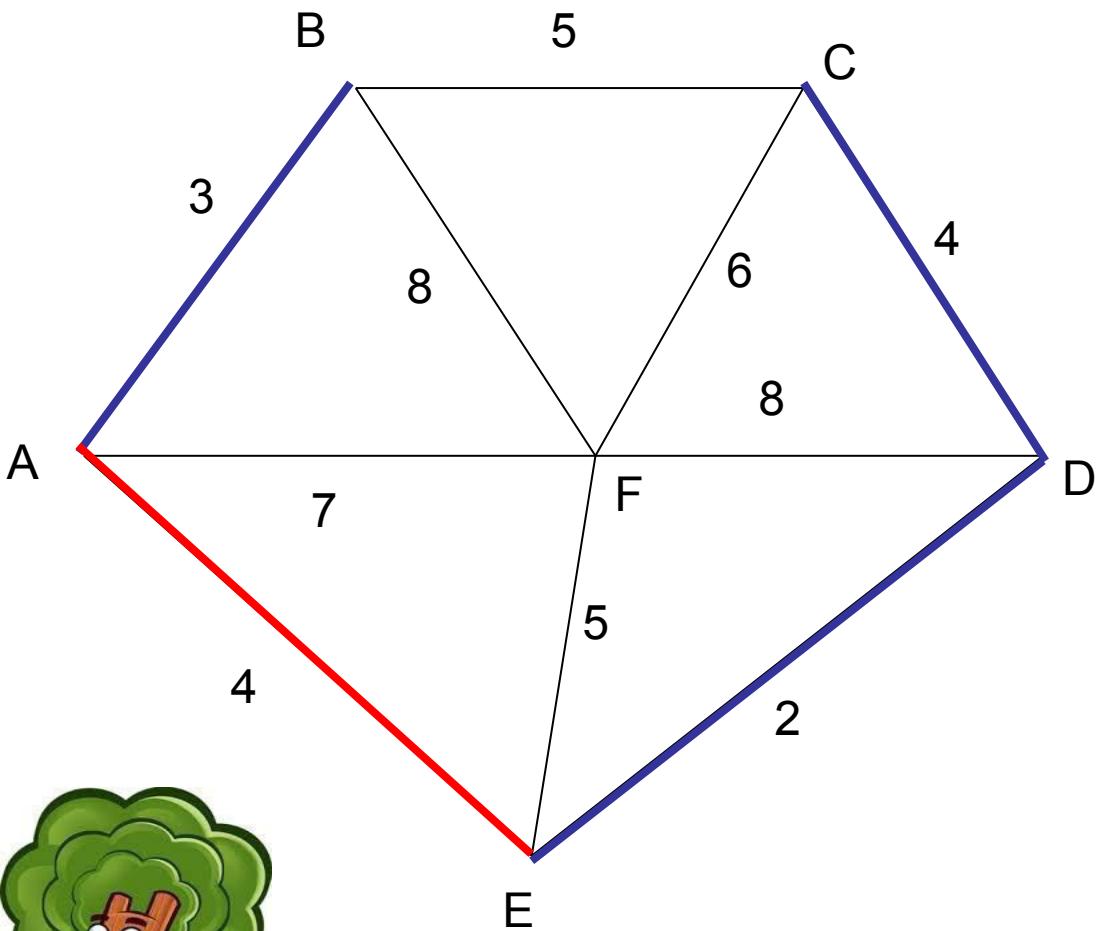
ED 2

AB 3

CD 4 (or AE 4)



Kruskal's Algorithm



Select the next shortest edge which does not create a cycle

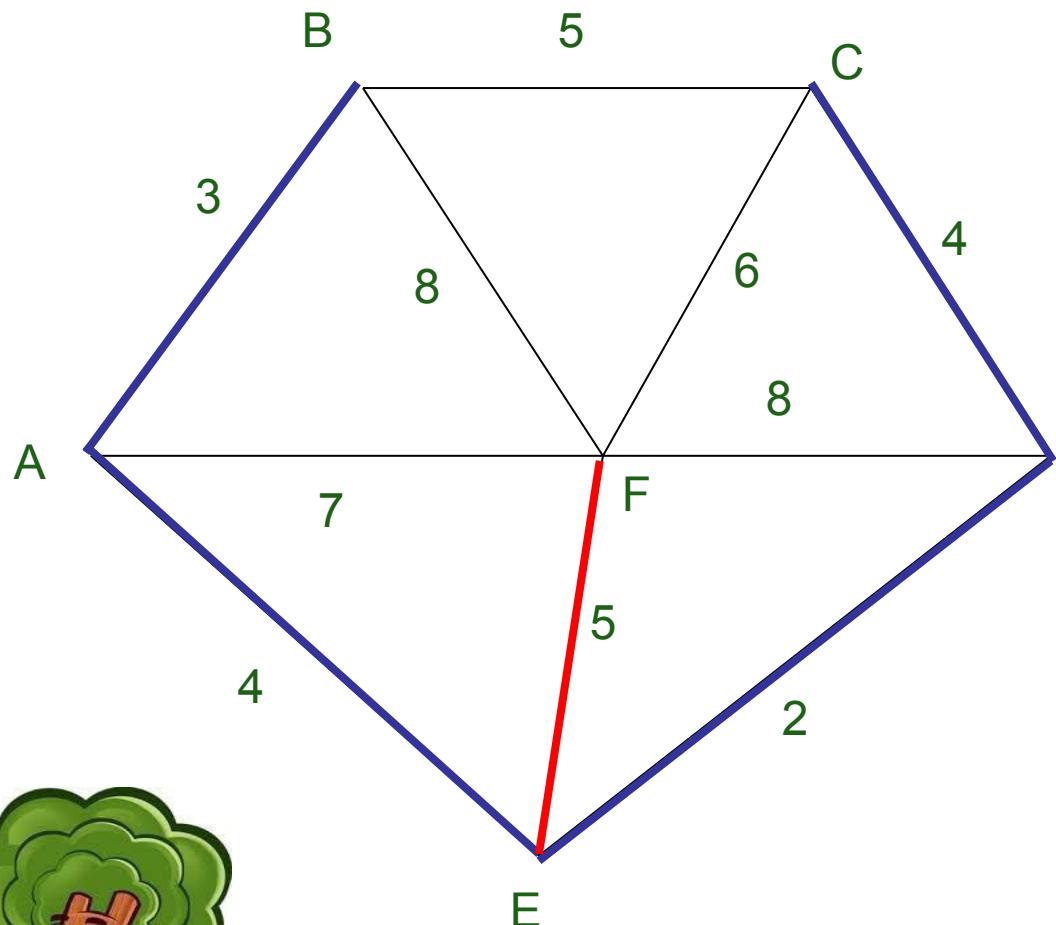
E

A

C



Kruskal's Algorithm



Select the next shortest edge which does not create a cycle

ED 2

AB 3

D CD 4

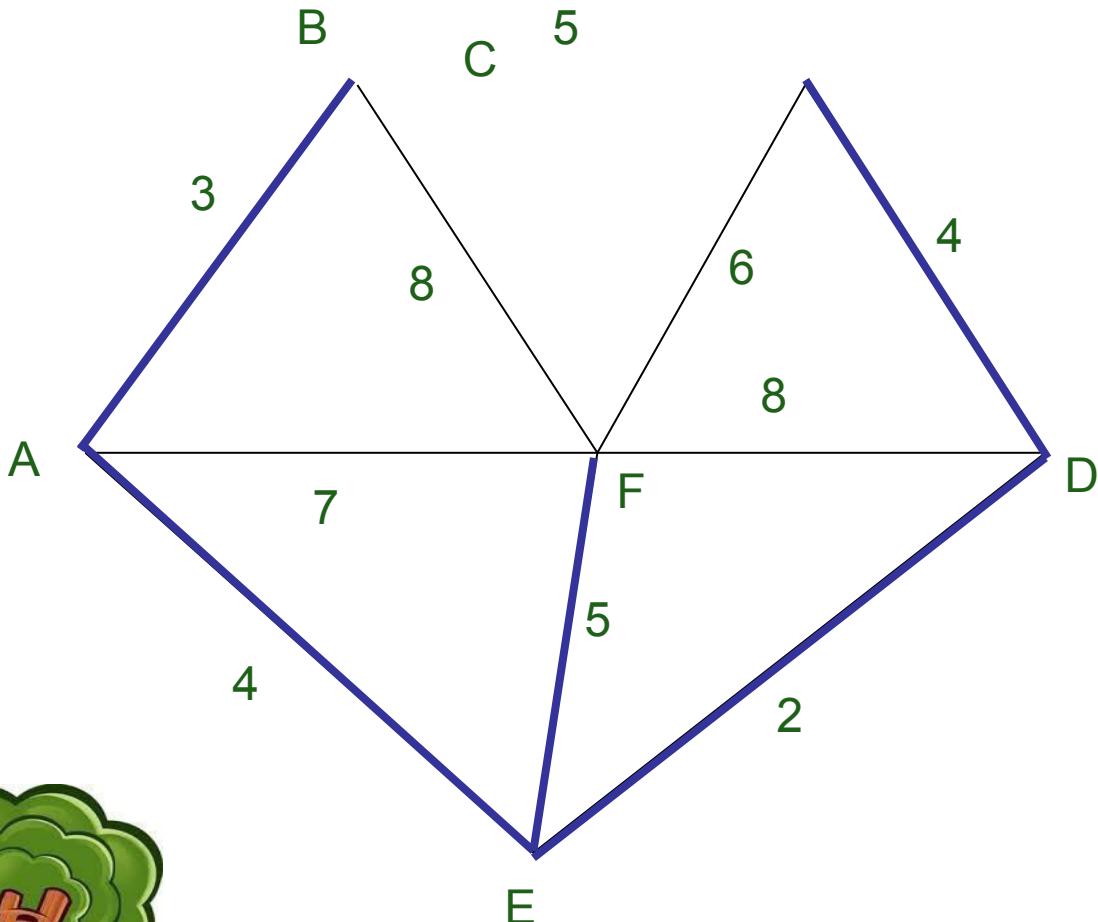
AE 4

BC 5 - forms a cycle

EF 5



Kruskal's Algorithm



All vertices have been connected.

The solution is

ED 2

AB 3

CD 4

AE 4

EF 5

Total weight of tree:
18



Algorithm

```
function Kruskal ( $G=(N,A)$ : graph ; length :  $A \subseteq \mathbb{R}^+$ ):set of edges
{initialisation}
sort A by increasing length
 $N \triangleq$  the number of nodes in N
 $T \triangleq \emptyset$  {will contain the edges of the minimum spanning tree}
initialise n sets, each containing the different element of N
{greedy loop}
repeat
     $e \triangleq \{u, v\} \triangleq$  shortest edge not yet considered
    ucomp  $\triangleq$  find( $u$ )
    vcomp  $\triangleq$  find( $v$ )
    if ucomp  $\neq$  vcomp then
        merge(ucomp , vcomp)
         $T \triangleq T \cup \{e\}$ 
until T contains  $n-1$  edges
return T
```



Kruskal's Algorithm: complexity

Sorting loop:

$O(a \log n)$

Initialization of components:

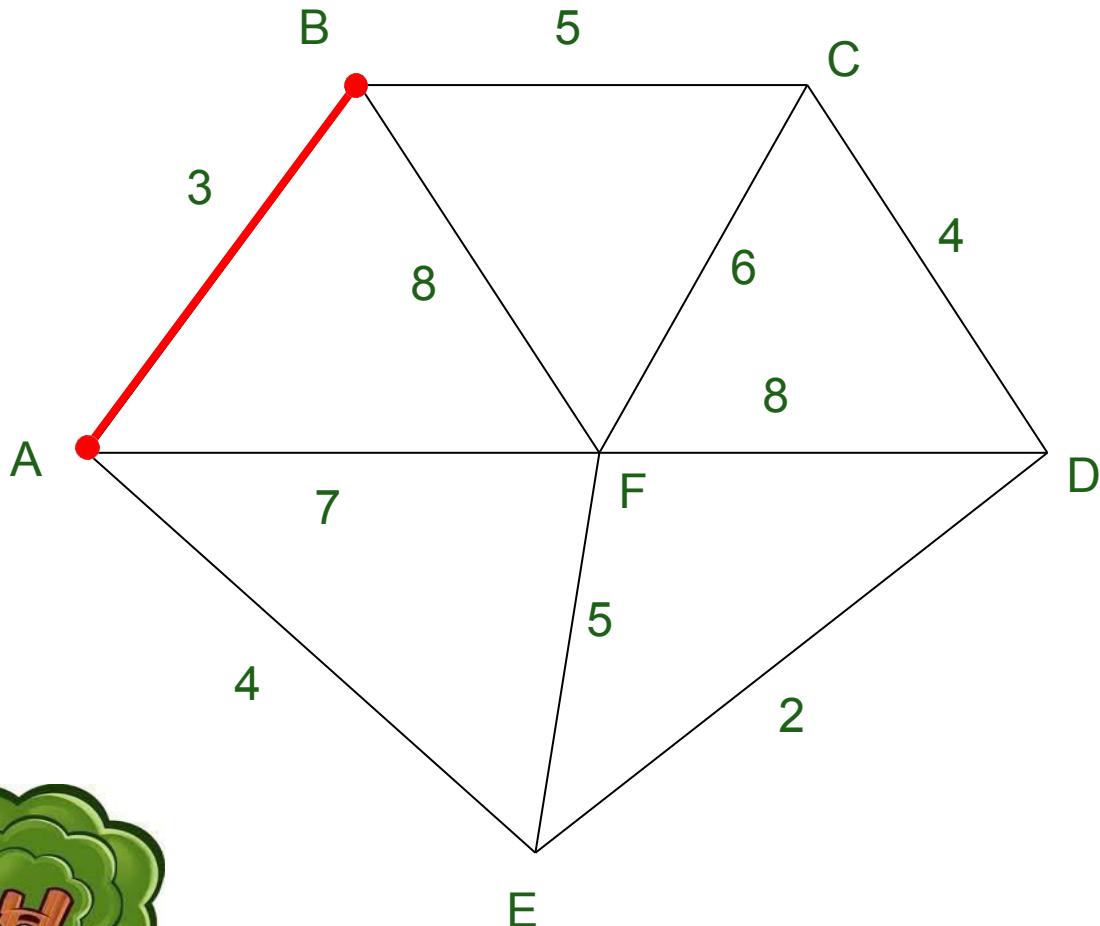
$O(n)$

Finding and merging:

$O(a \log n)$

$O(a \log n)$

Prim's Algorithm



Select any vertex

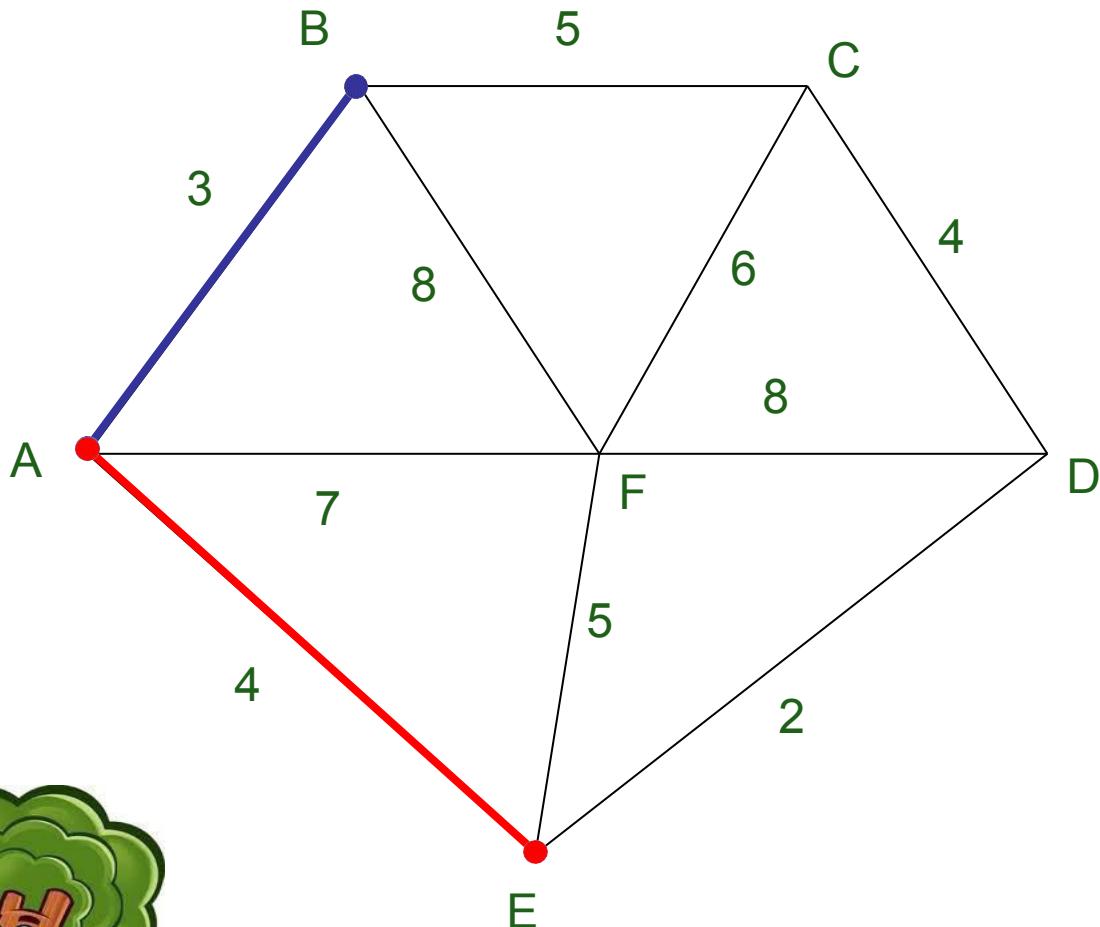
A

Select the shortest edge connected to that vertex

AB 3



Prim's Algorithm

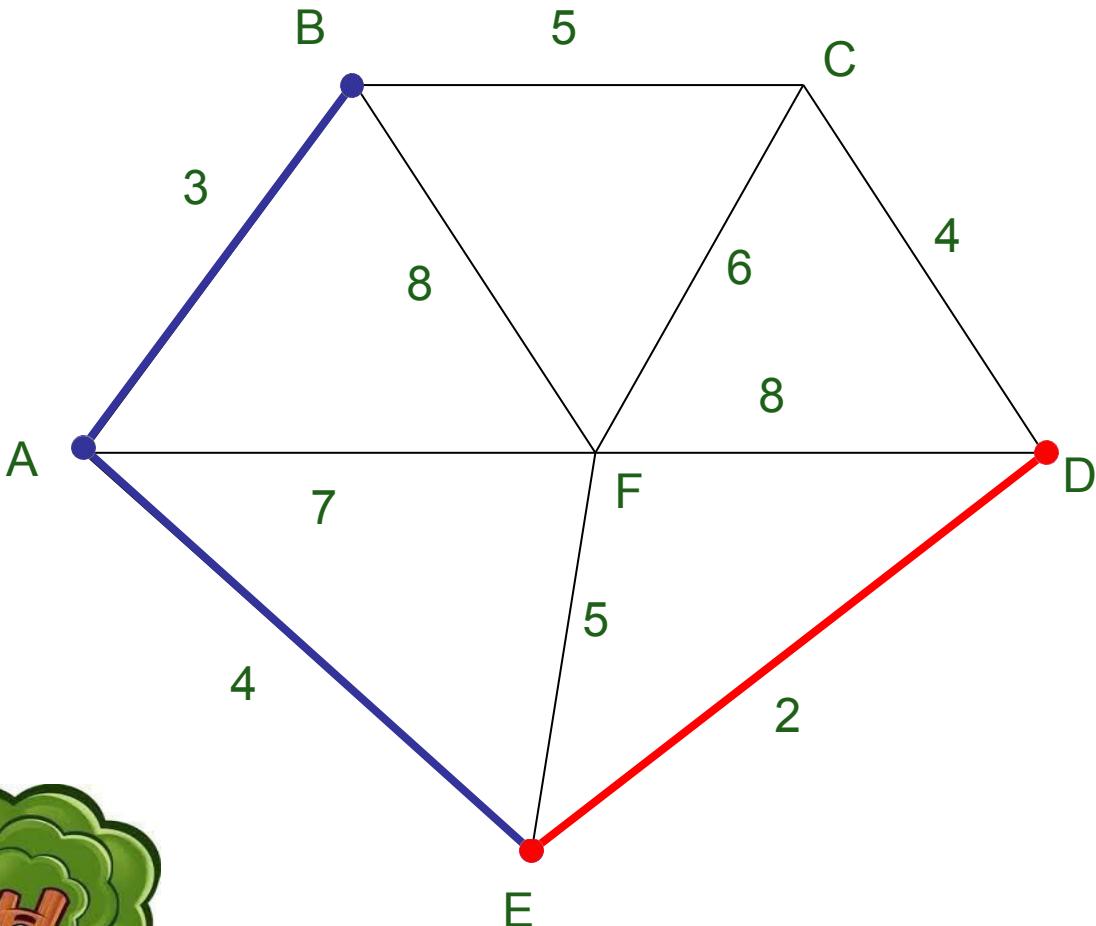


Select the shortest edge connected to any vertex already connected.

AE 4



Prim's Algorithm

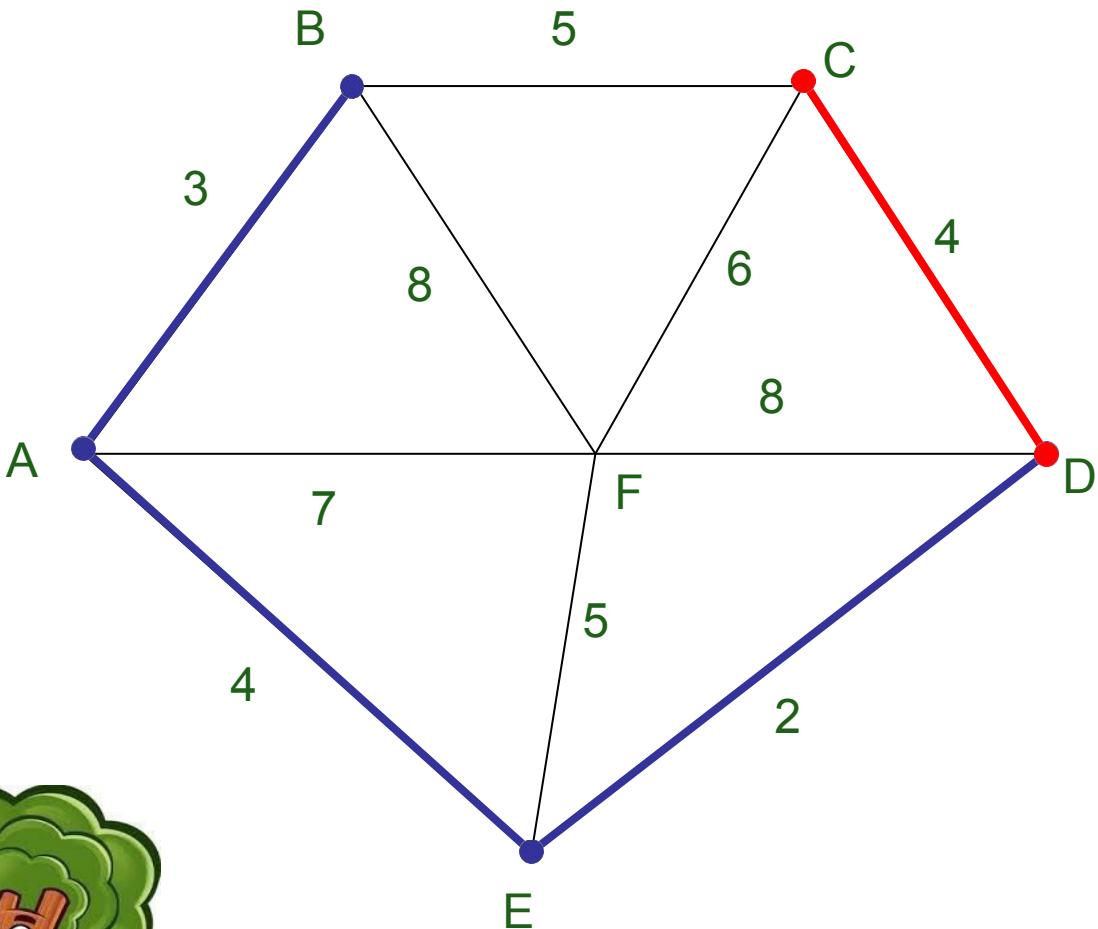


Select the shortest edge connected to any vertex already connected.

ED 2



Prim's Algorithm

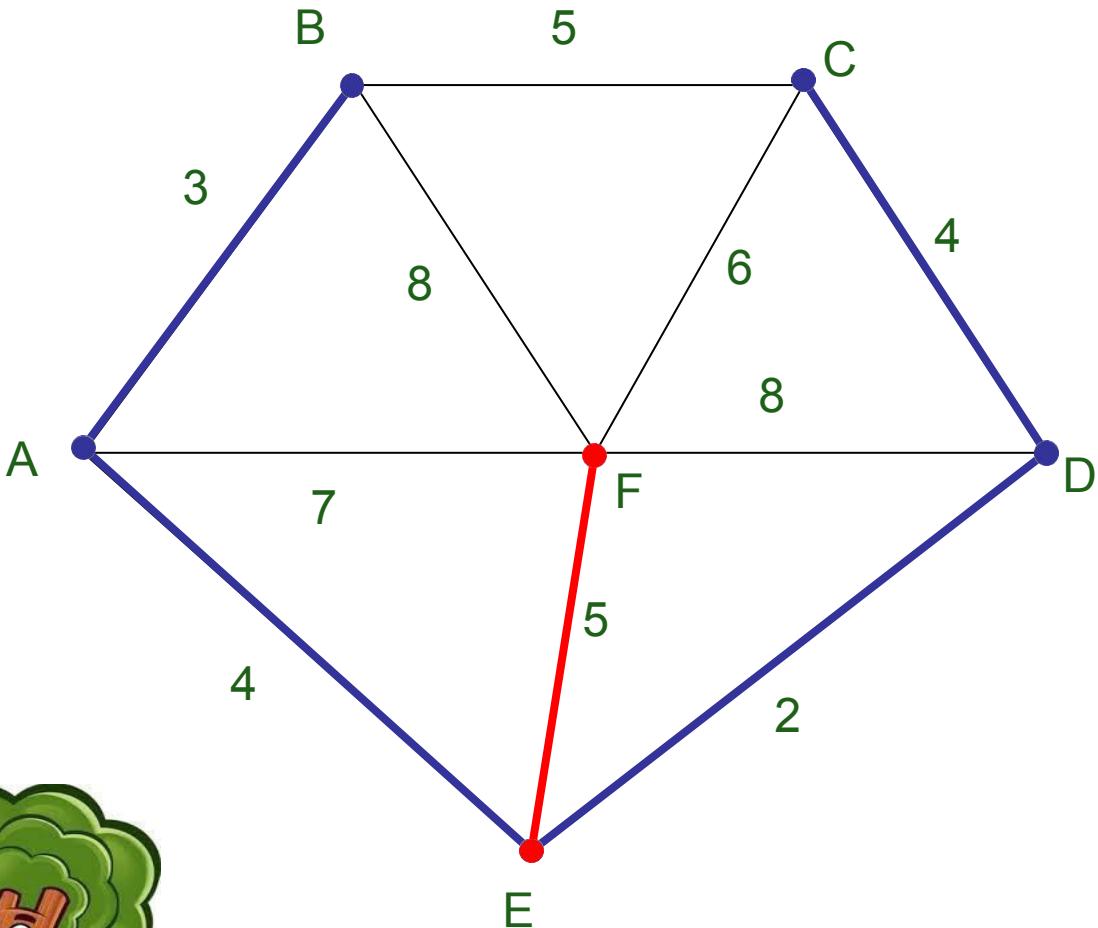


Select the shortest edge connected to any vertex already connected.

DC 4



Prim's Algorithm

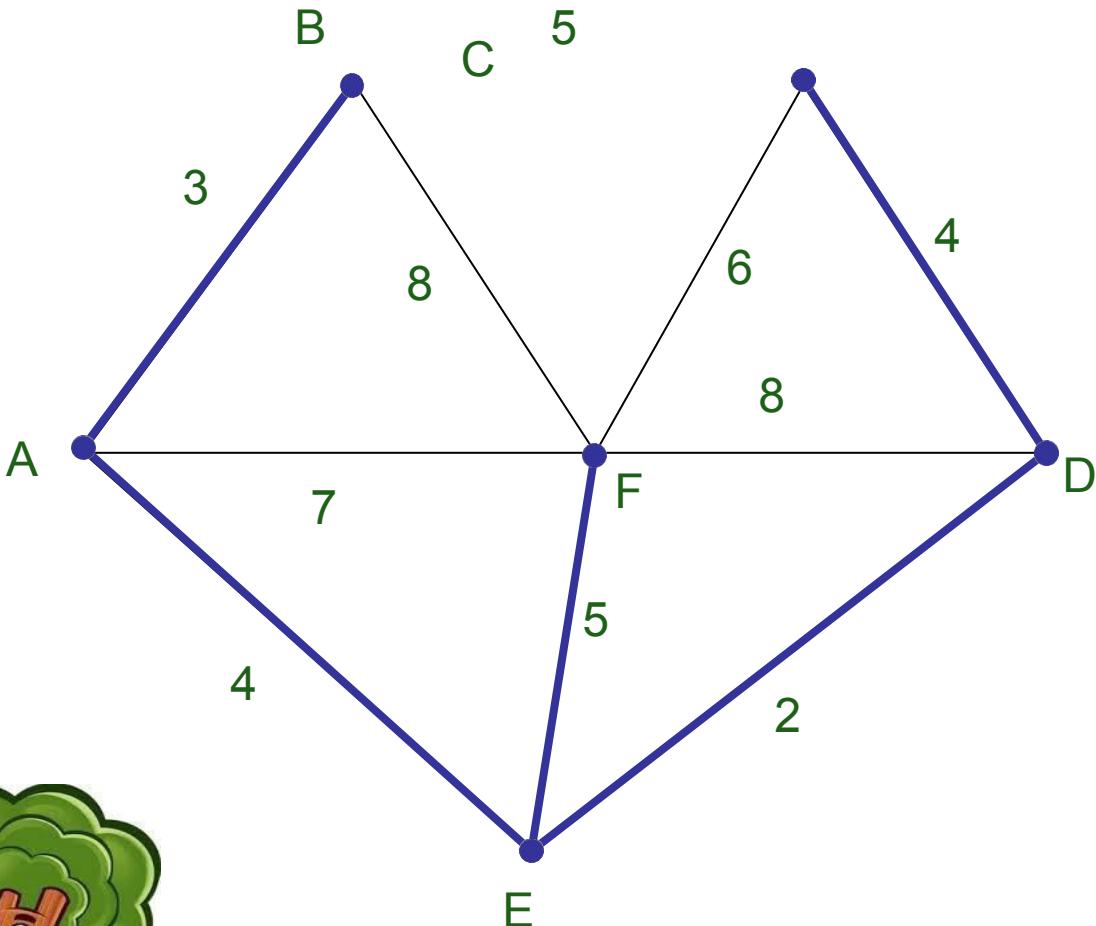


Select the shortest edge connected to any vertex already connected.

EF 5



Prim's Algorithm



All vertices have been connected.

The solution is

AB 3

AE 4

ED 2

DC 4

EF 5

Total weight of tree:
18



Prim's Algorithm

function Prim($G = \langle N, A \rangle$: graph ; length : $A \rightarrow \mathbb{R}^+$) : set of edges

{initialisation}

$T \leftarrow \emptyset$

$B \leftarrow \{ \text{an arbitrary member of } N \}$

While $B \neq N$ do

 find $e = \{u, v\}$ of minimum length such
 $u \in B$ and $v \in N \setminus B$

$T \leftarrow T \cup \{e\}$

$B \leftarrow B \cup \{v\}$

Return T

Complexity:

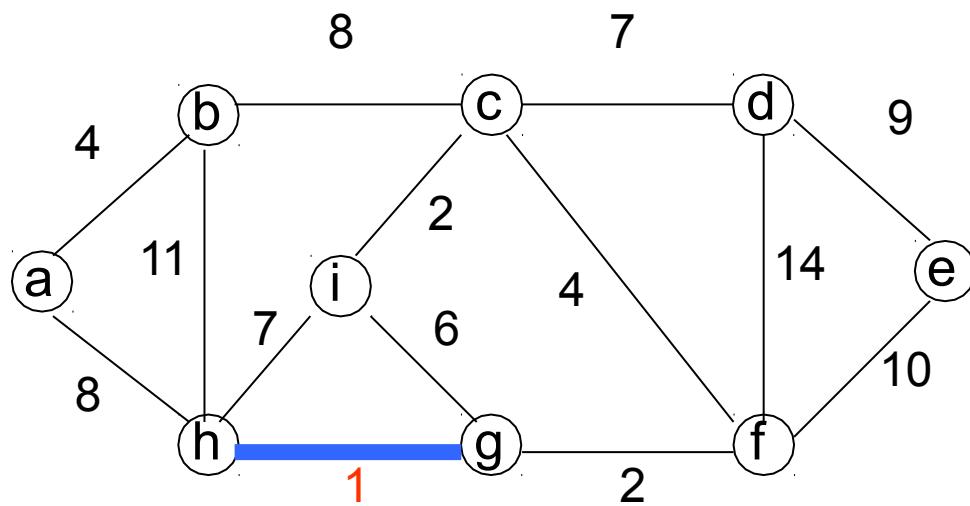
Outer loop: $n-1$ times

Inner loop: n times

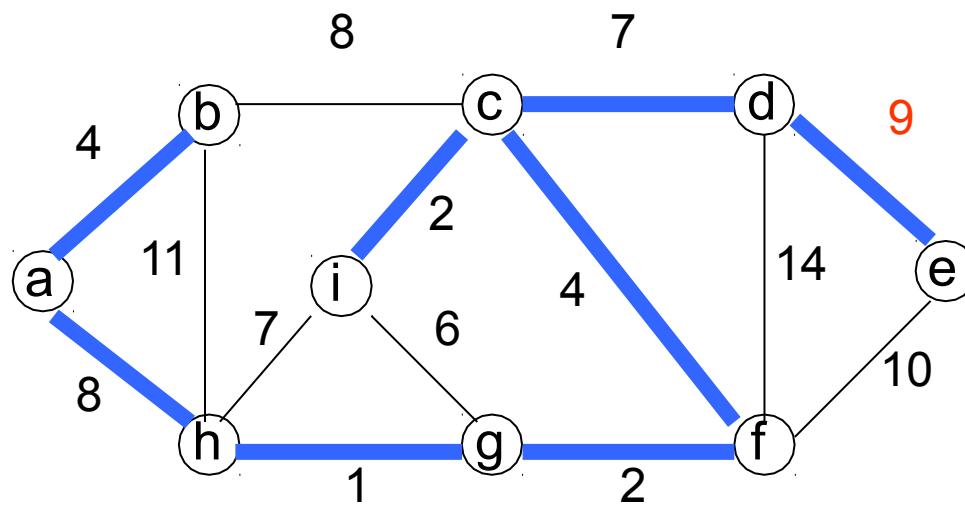
$O(n^2)$



Example



Solution



Minimum Connector Algorithms

Kruskal's algorithm

1. Select the shortest edge in a network
2. Select the next shortest edge which does not create a cycle
3. Repeat step 2 until all vertices have been connected



Prim's algorithm

1. Select any vertex
2. Select the shortest edge connected to that vertex
3. Select the shortest edge connected to any vertex already connected
4. Repeat step 3 until all vertices have been connected



Thank you!!

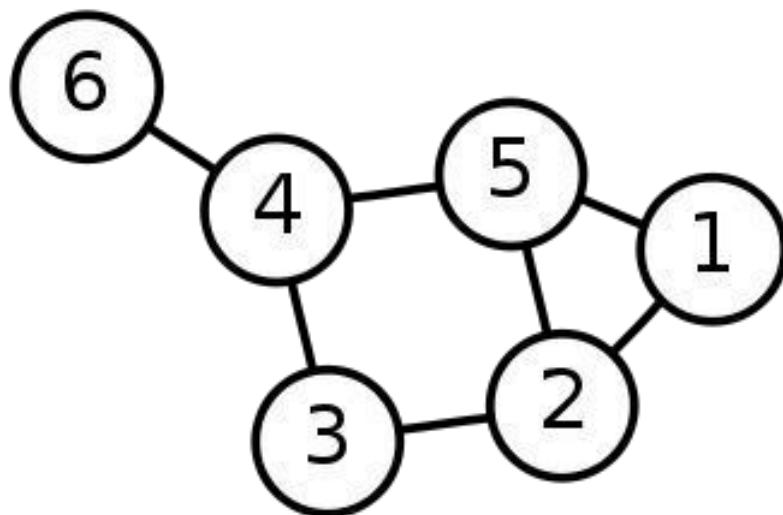




Dijkstra's Algorithm

Single-Source Shortest Path Problem

Single-Source Shortest Path Problem - The problem of finding shortest paths from a source vertex v to all other vertices in the graph.



Dijkstra's algorithm

Dijkstra's algorithm - is a solution to the single-source shortest path problem in graph theory.

Works on both directed and undirected graphs. However, all edges must have nonnegative weights.

Input: Weighted graph $G=\{E,V\}$ and source vertex $v \in V$, such that all edge weights are nonnegative

Output: Lengths of shortest paths (or the shortest paths themselves) from a given source vertex $v \in V$ to all other vertices

Approach

- The algorithm computes for each vertex u the **distance** to u from the start vertex v , that is, the weight of a shortest path between v and u .
- the algorithm keeps track of the set of vertices for which the distance has been computed, called the **cloud** C
- Every vertex has a label D associated with it. For any vertex u , $D[u]$ stores an approximation of the distance between v and u . The algorithm will update a $D[u]$ value when it finds a shorter path from v to u .
- When a vertex u is added to the cloud, its label $D[u]$ is equal to the actual (final) distance between the starting vertex v and vertex u .

Dijkstra pseudocode

Dijkstra(v_1, v_2):

for each vertex v : // Initialization

v 's distance := infinity.

v 's previous := none.

v_1 's distance := 0.

List := {all vertices}.

while List is not empty:

v := remove List vertex with minimum distance.

mark v as known.

for each unknown neighbor n of v :

$dist$:= v 's distance + edge (v, n) 's weight.

if $dist$ is smaller than n 's distance:

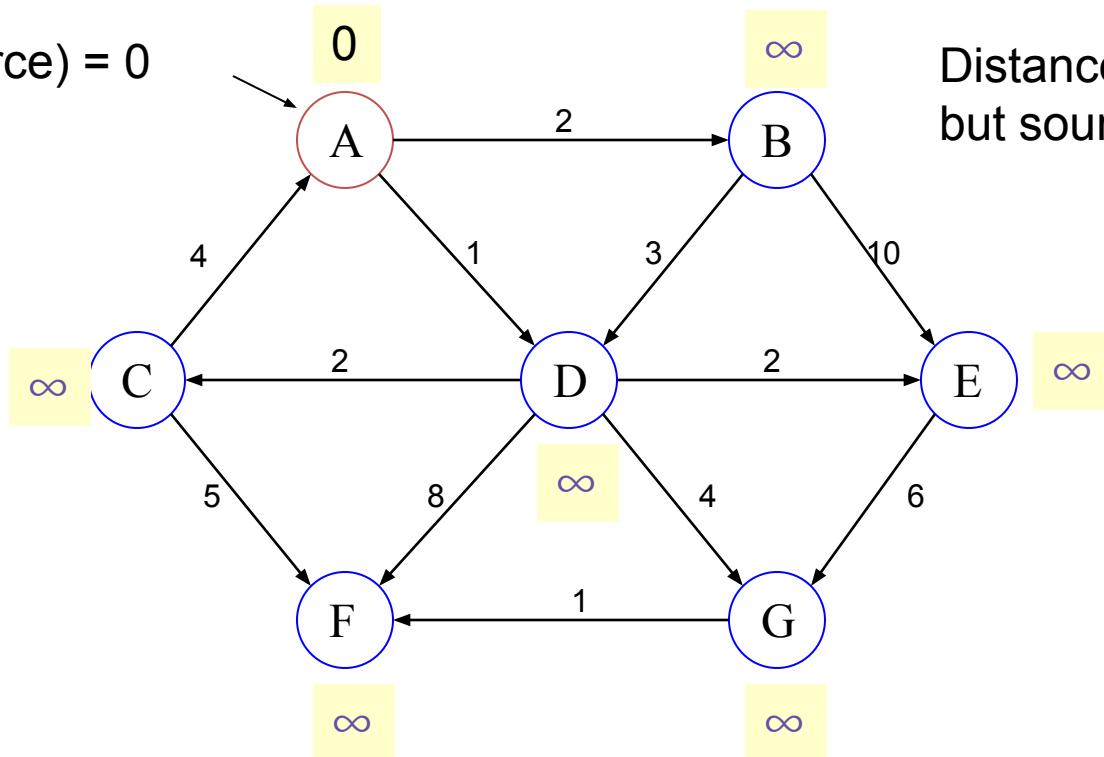
n 's distance := $dist$.

n 's previous := v .

*reconstruct path from v_2 back to v_1 ,
following previous pointers.*

Example: Initialization

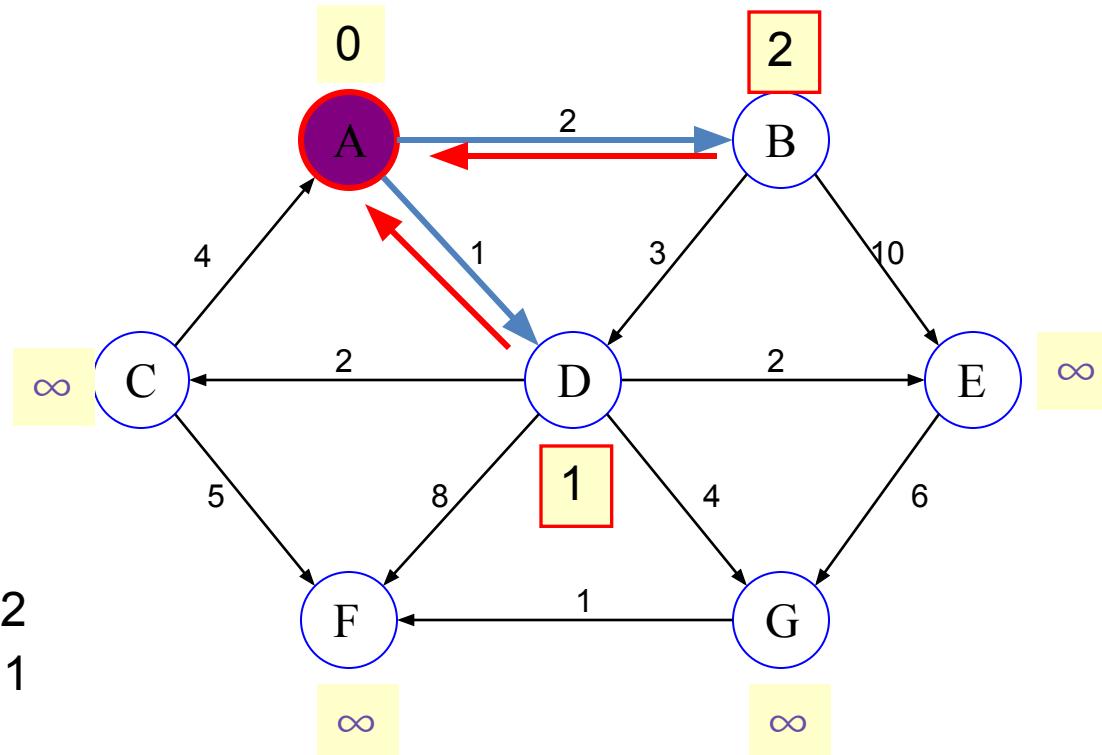
Distance(source) = 0



Distance (all vertices but source) = ∞

Pick vertex in List with minimum distance.

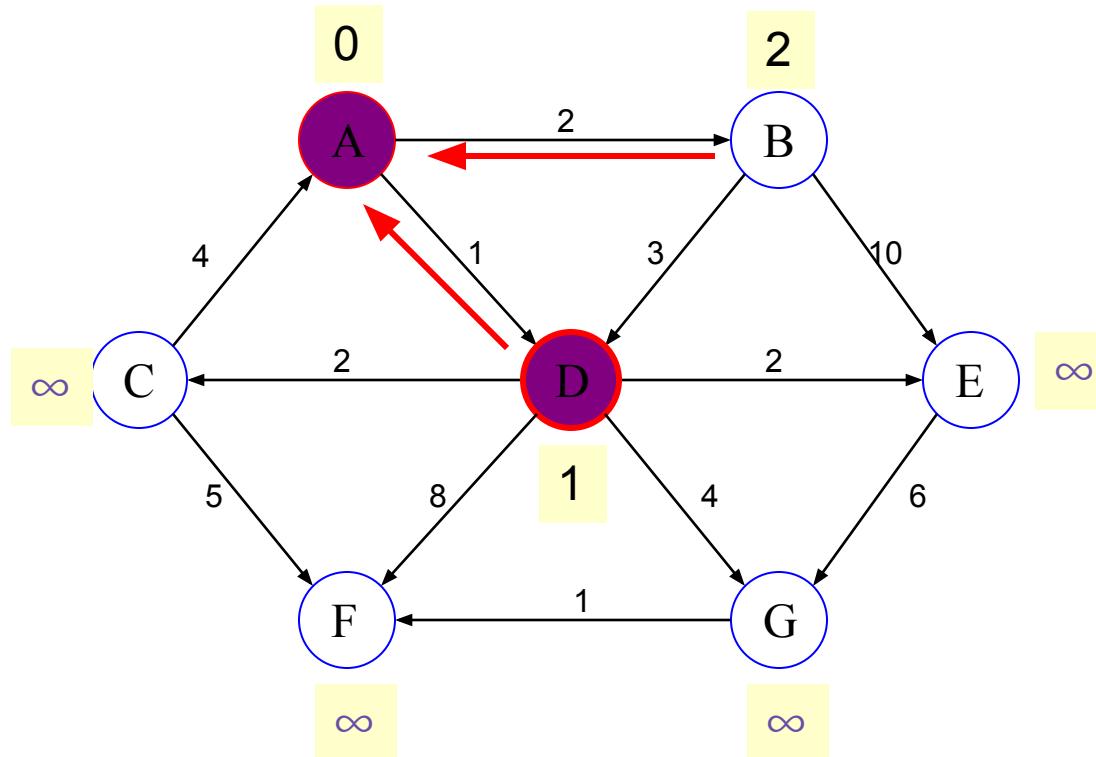
Example: Update neighbors' distance



$$\text{Distance}(B) = 2$$

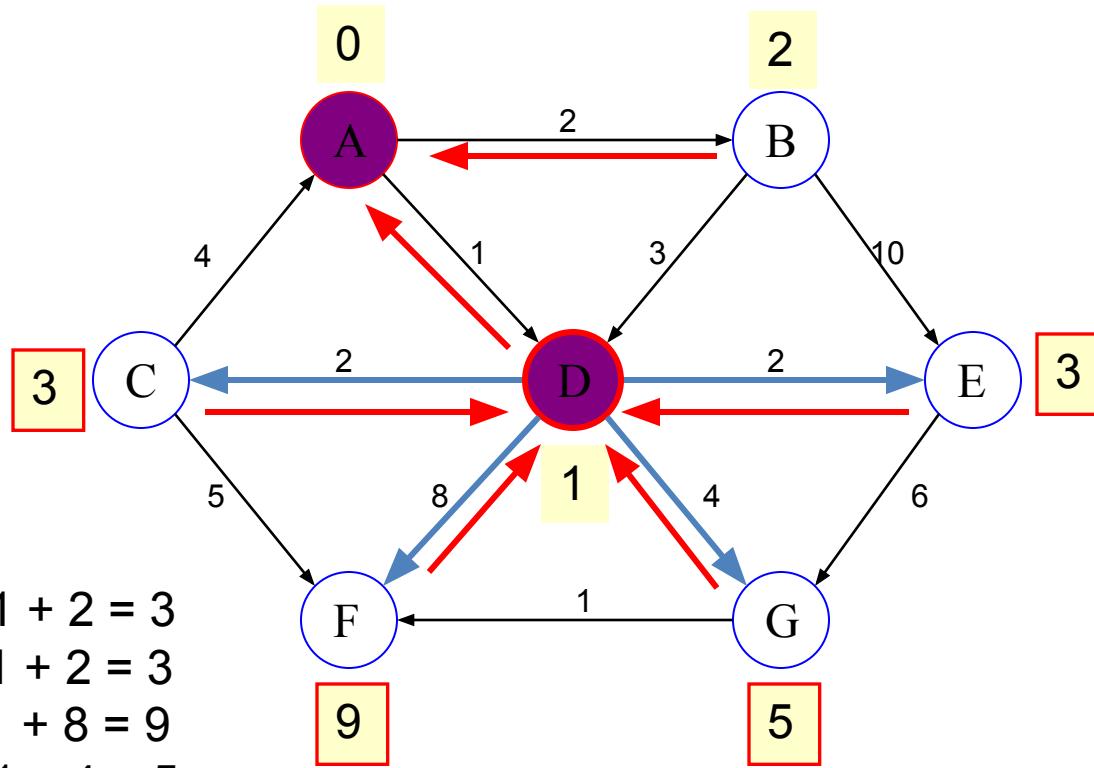
$$\text{Distance}(D) = 1$$

Example: Remove vertex with minimum distance



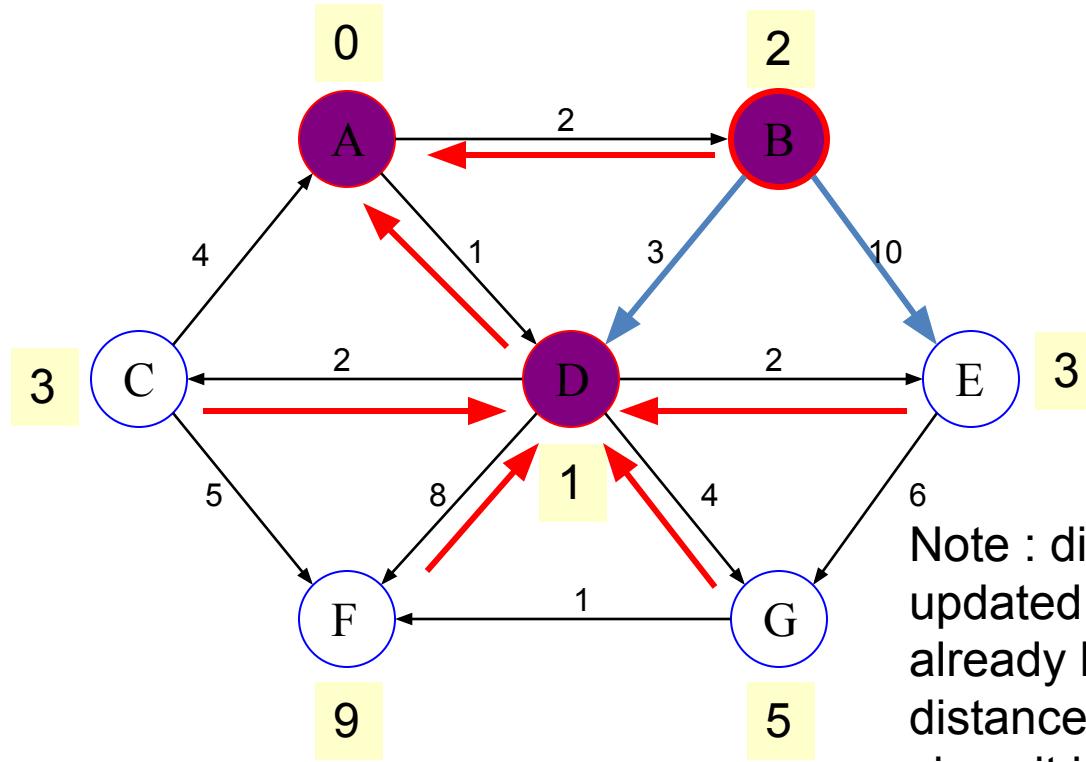
Pick vertex in List with minimum distance, i.e., D

Example: Update neighbors



Example: Continued...

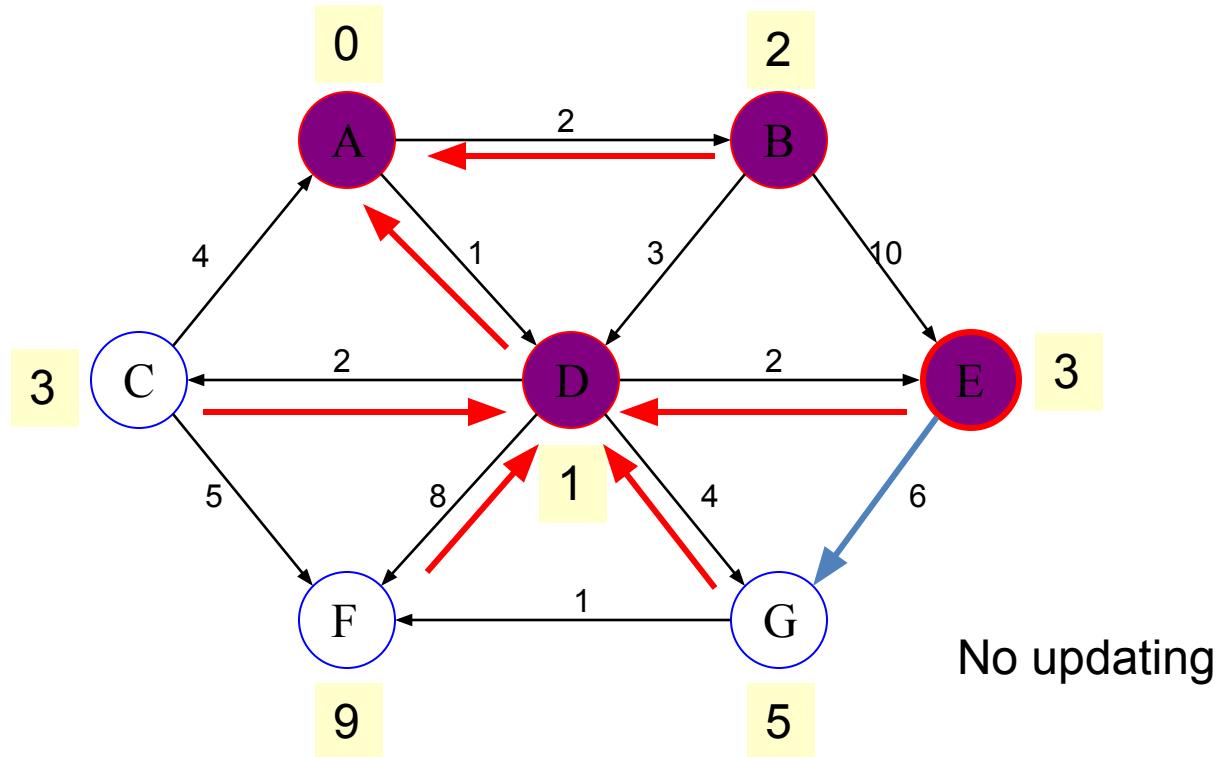
Pick vertex in List with minimum distance (B) and update neighbors



Note : distance(D) not updated since D is already known and distance(E) not updated since it is larger than previously computed

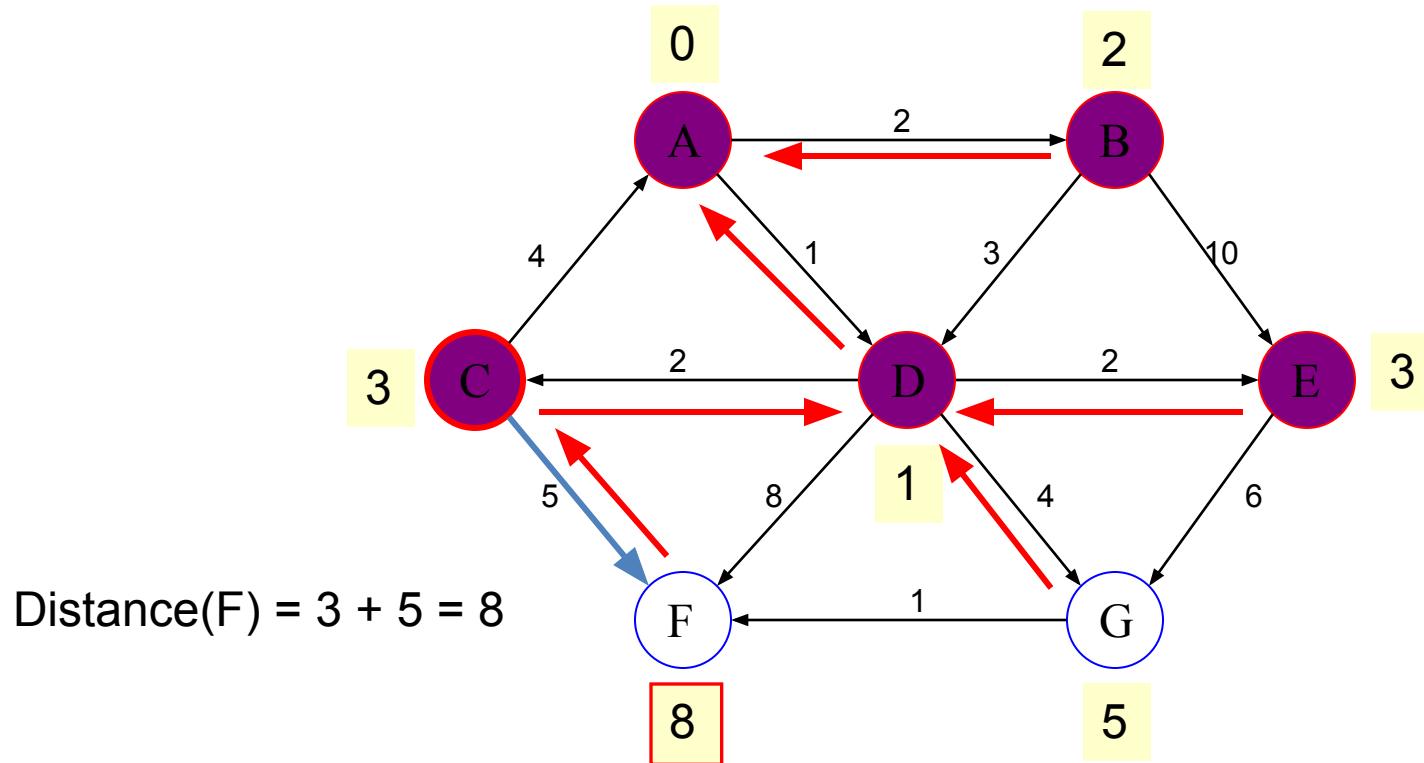
Example: Continued...

Pick vertex List with minimum distance (E) and update neighbors



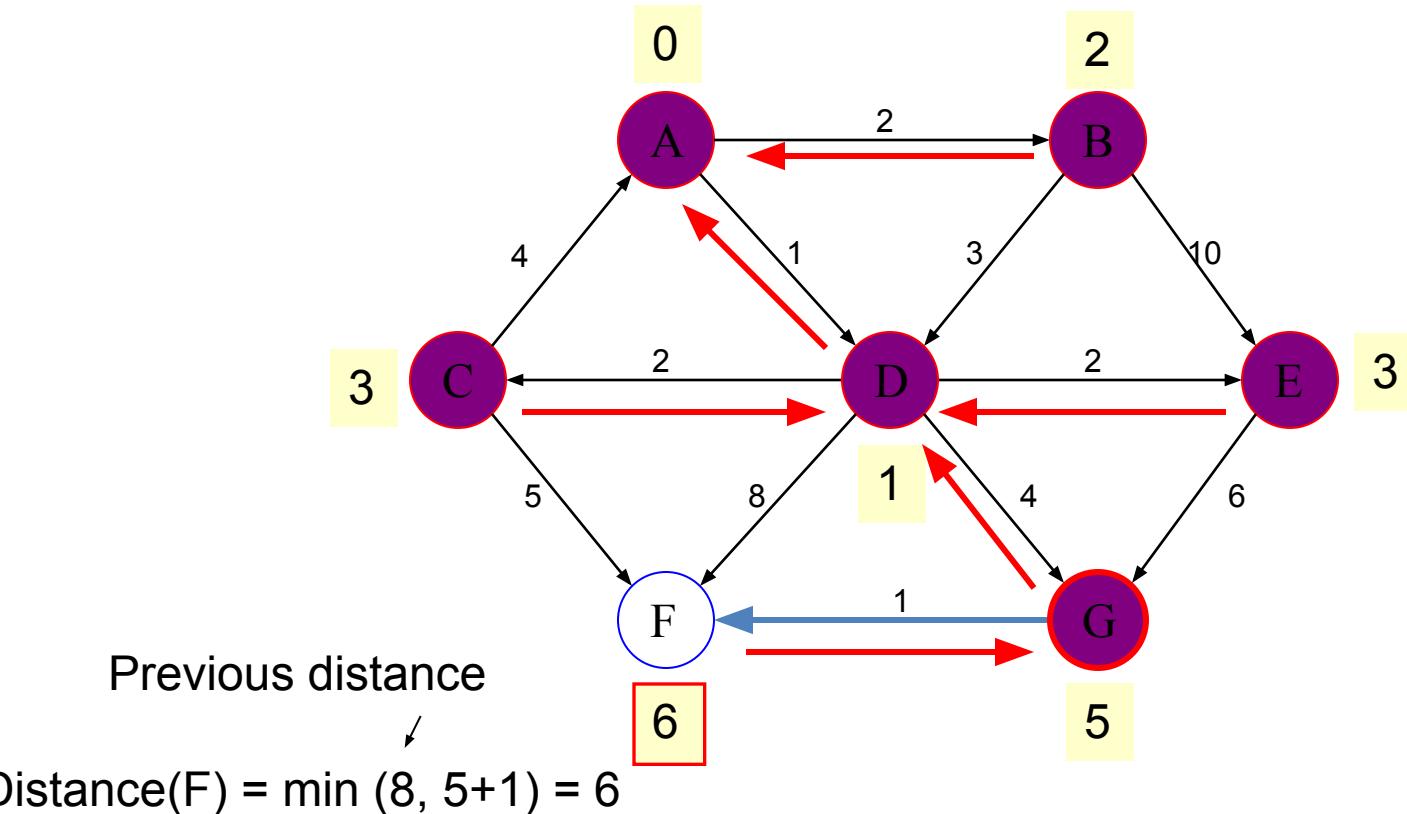
Example: Continued...

Pick vertex List with minimum distance (C) and update neighbors

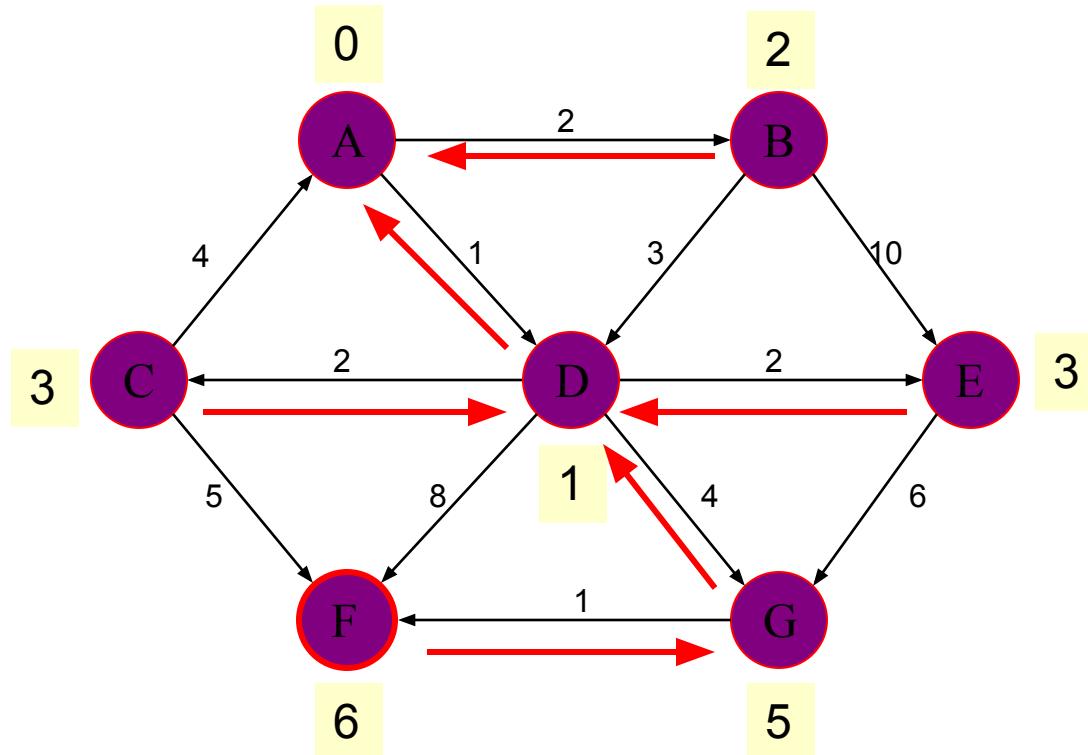


Example: Continued...

Pick vertex List with minimum distance (G) and update neighbors

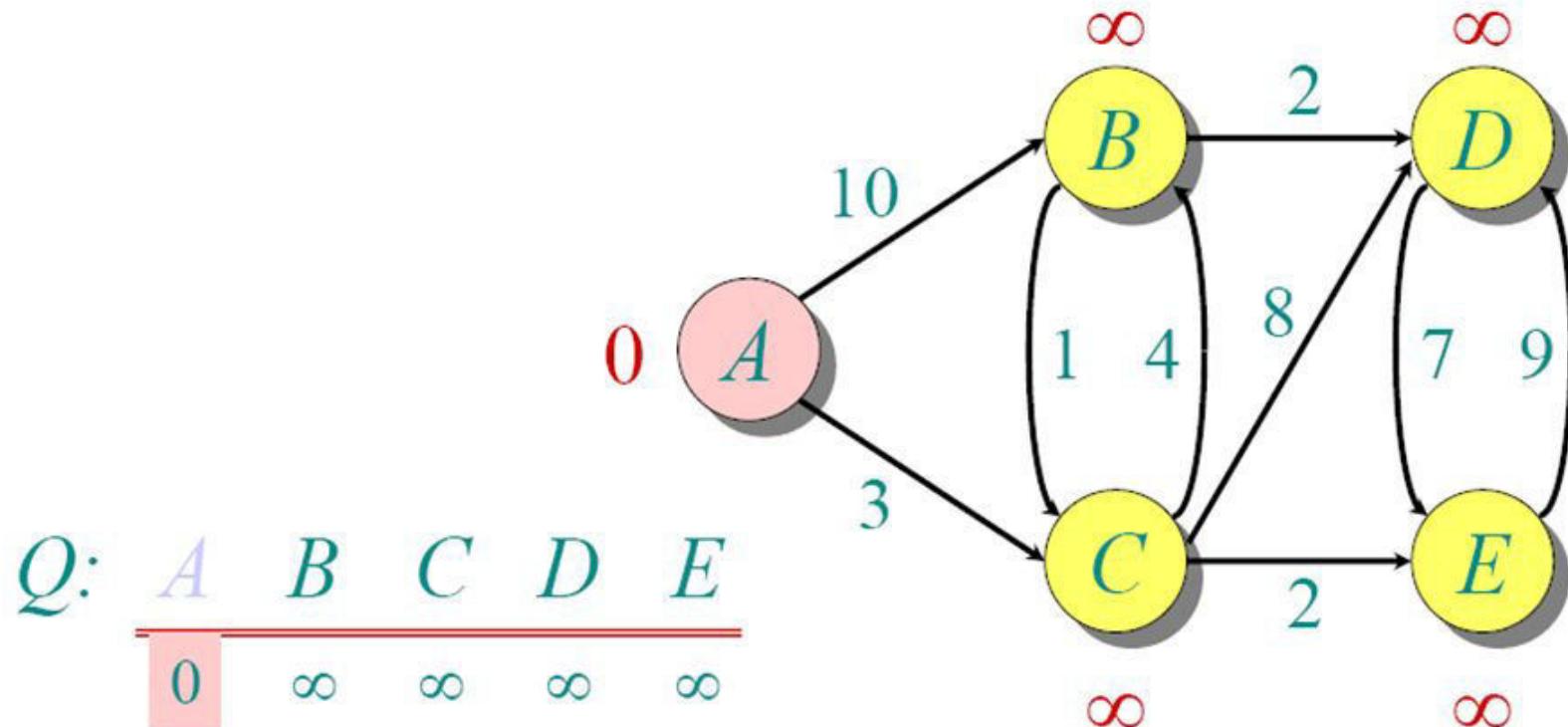


Example (end)

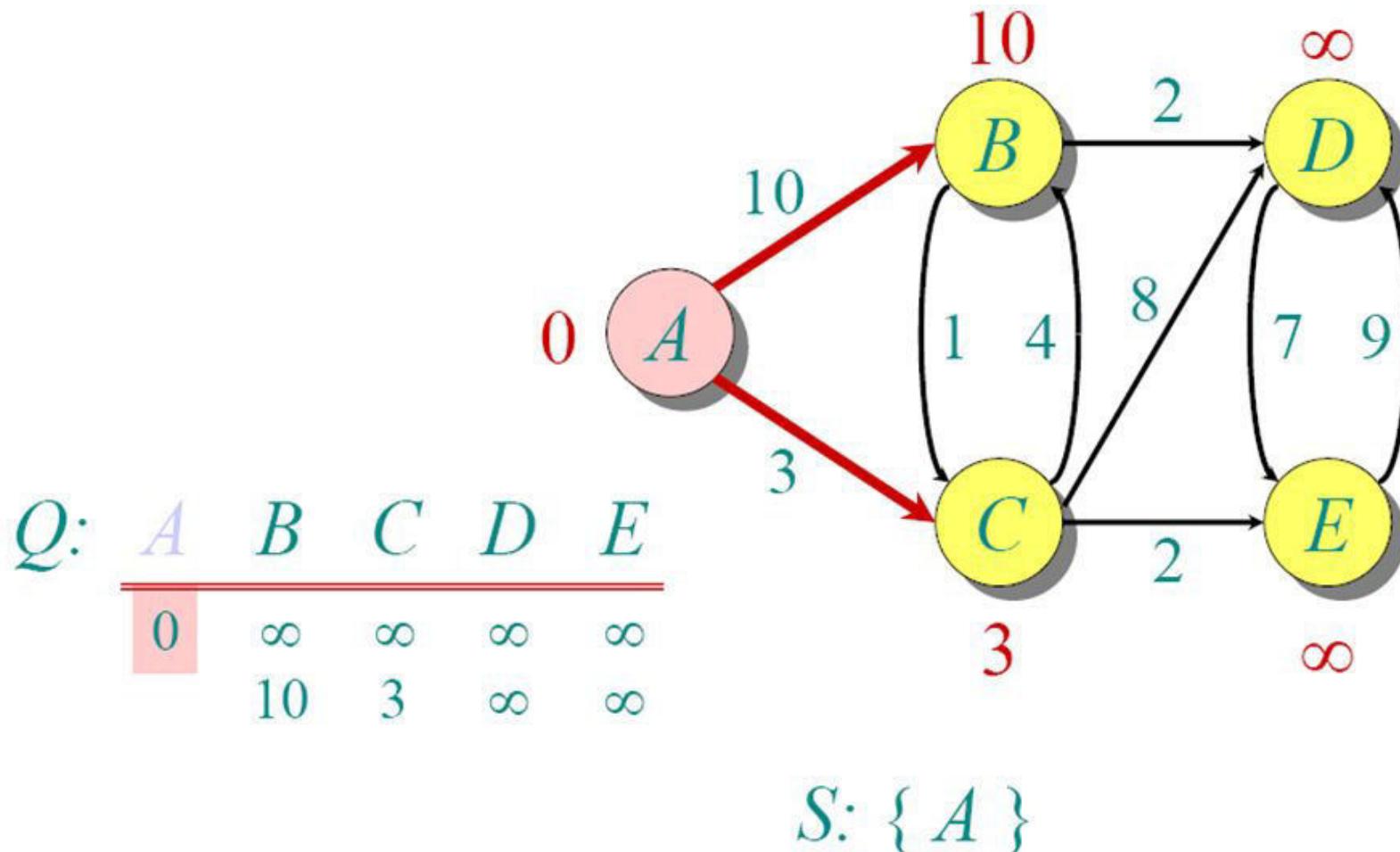


Pick vertex not in S with lowest cost (F) and update neighbors

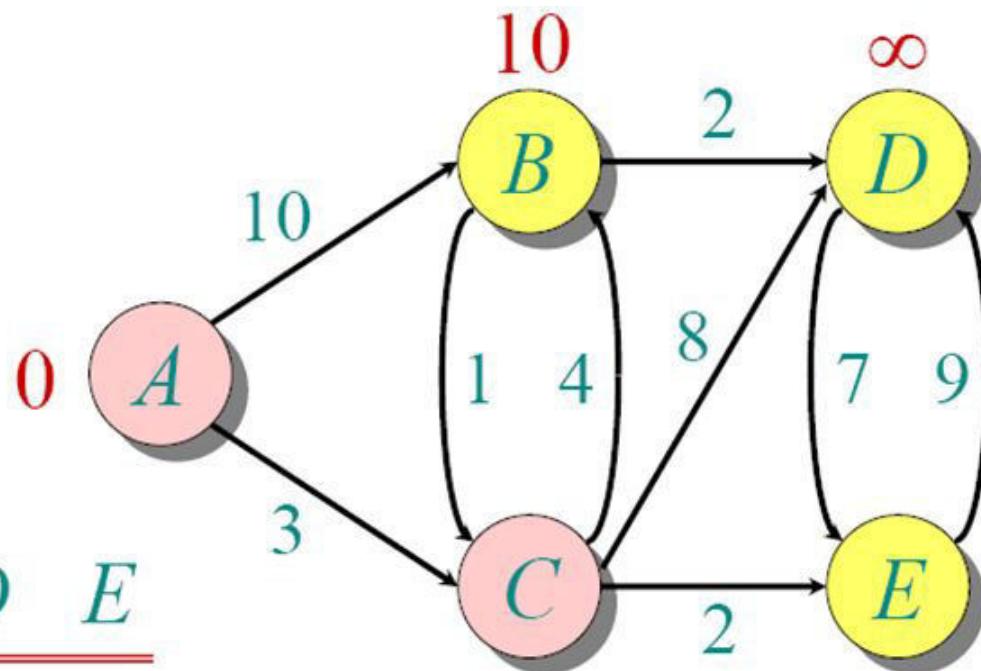
Another Example



Another Example



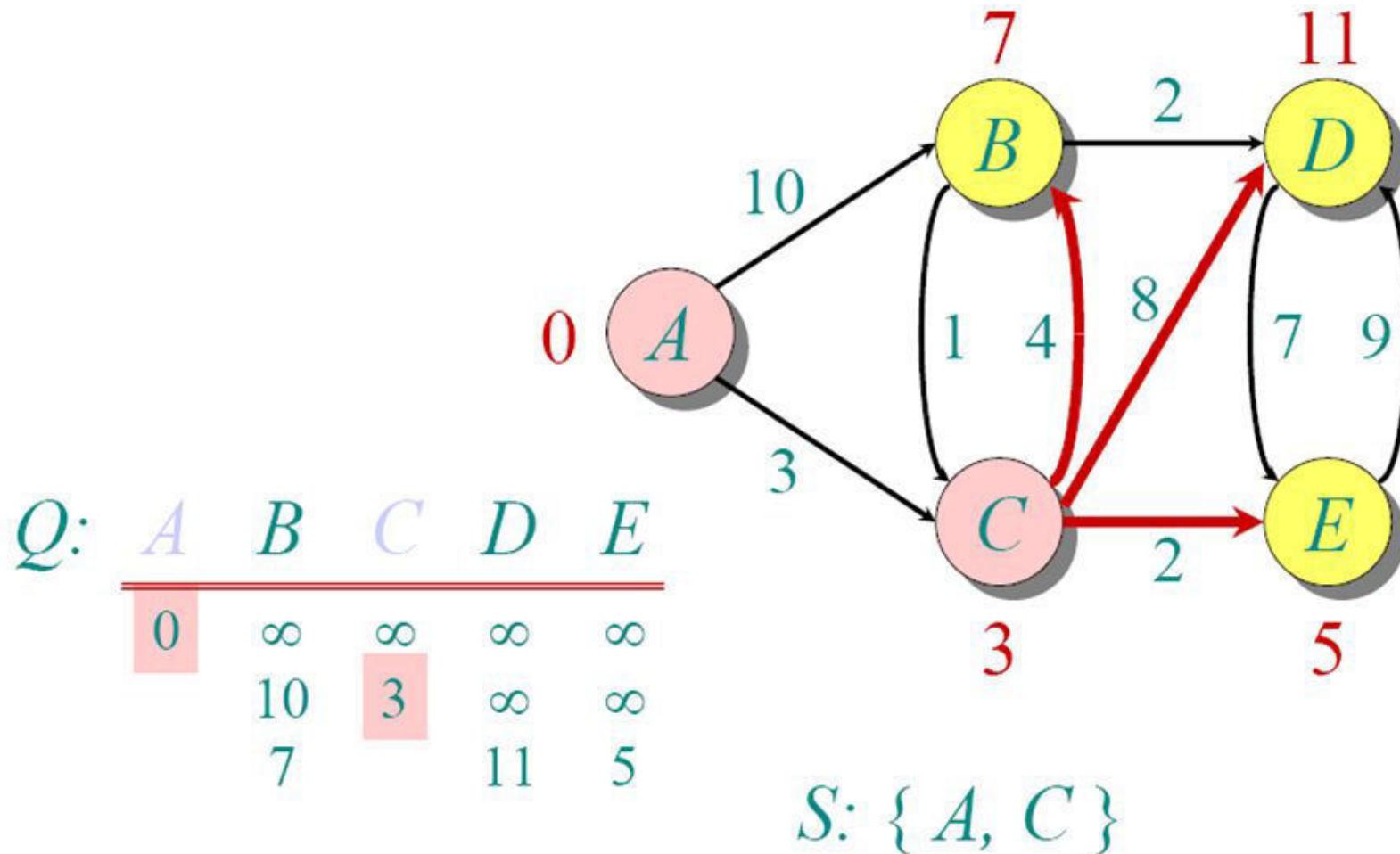
Another Example



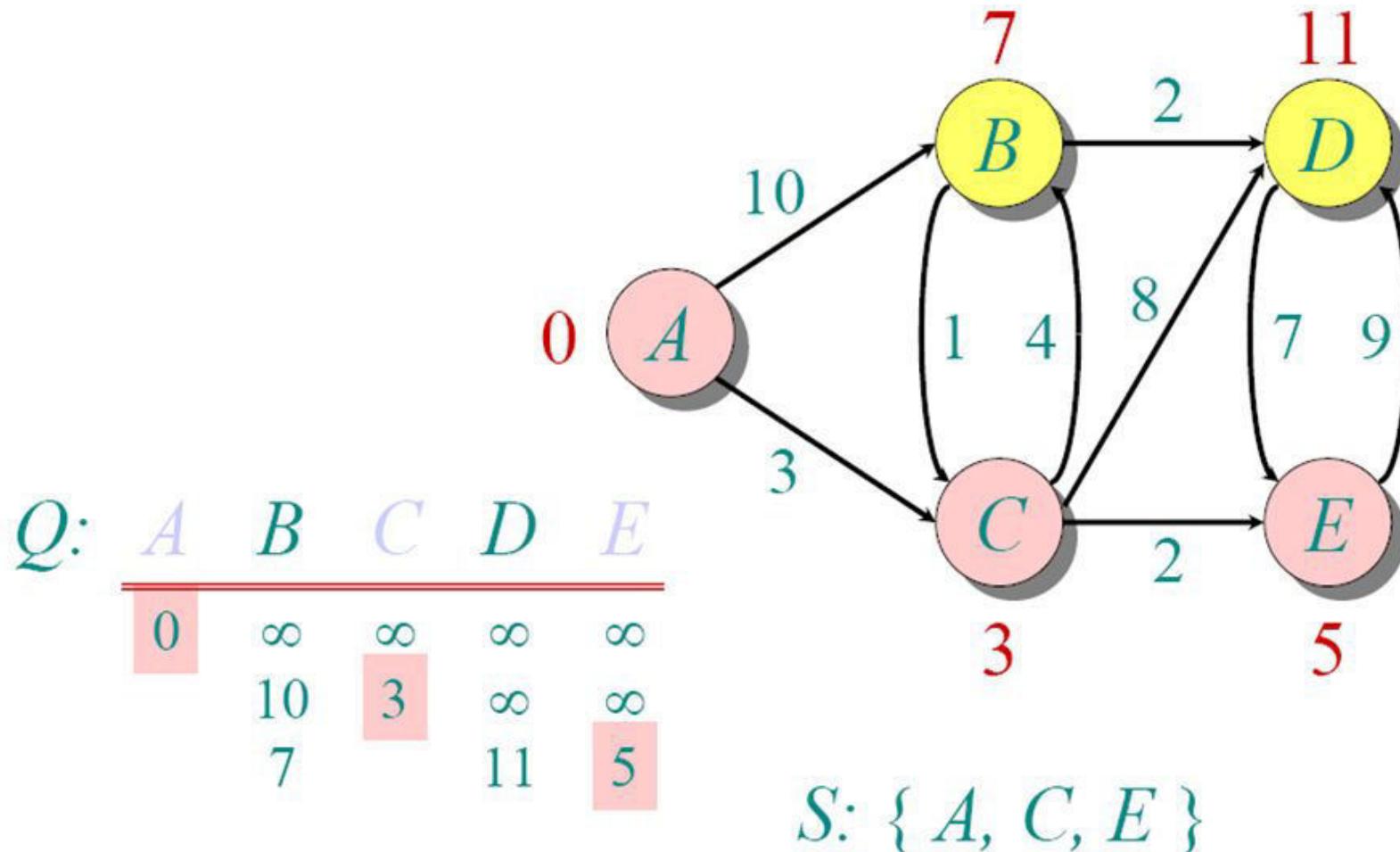
$Q:$	A	B	C	D	E
0	∞	∞	∞	∞	∞
10	3	∞	∞	∞	∞

$S: \{ A, C \}$

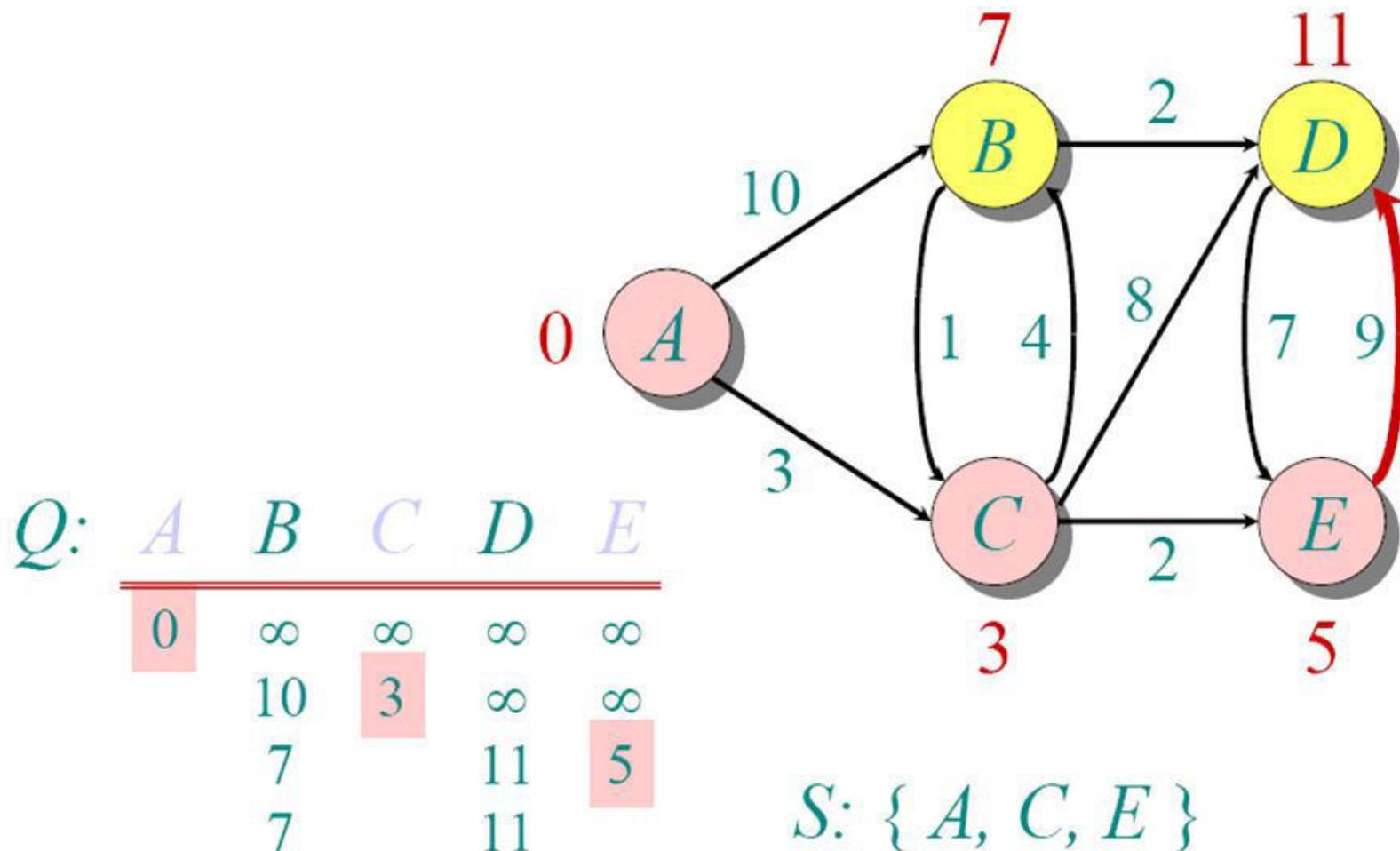
Another Example



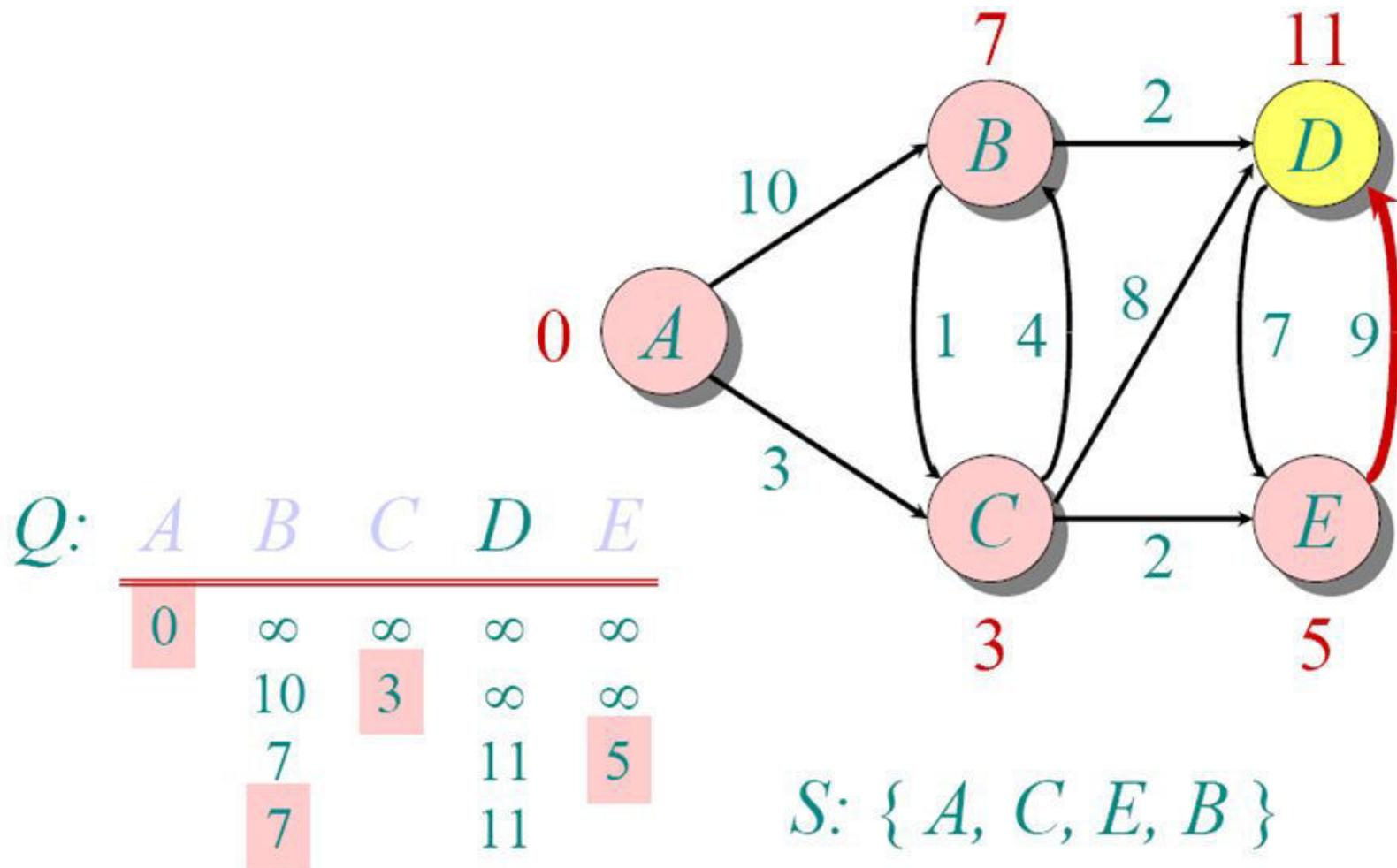
Another Example



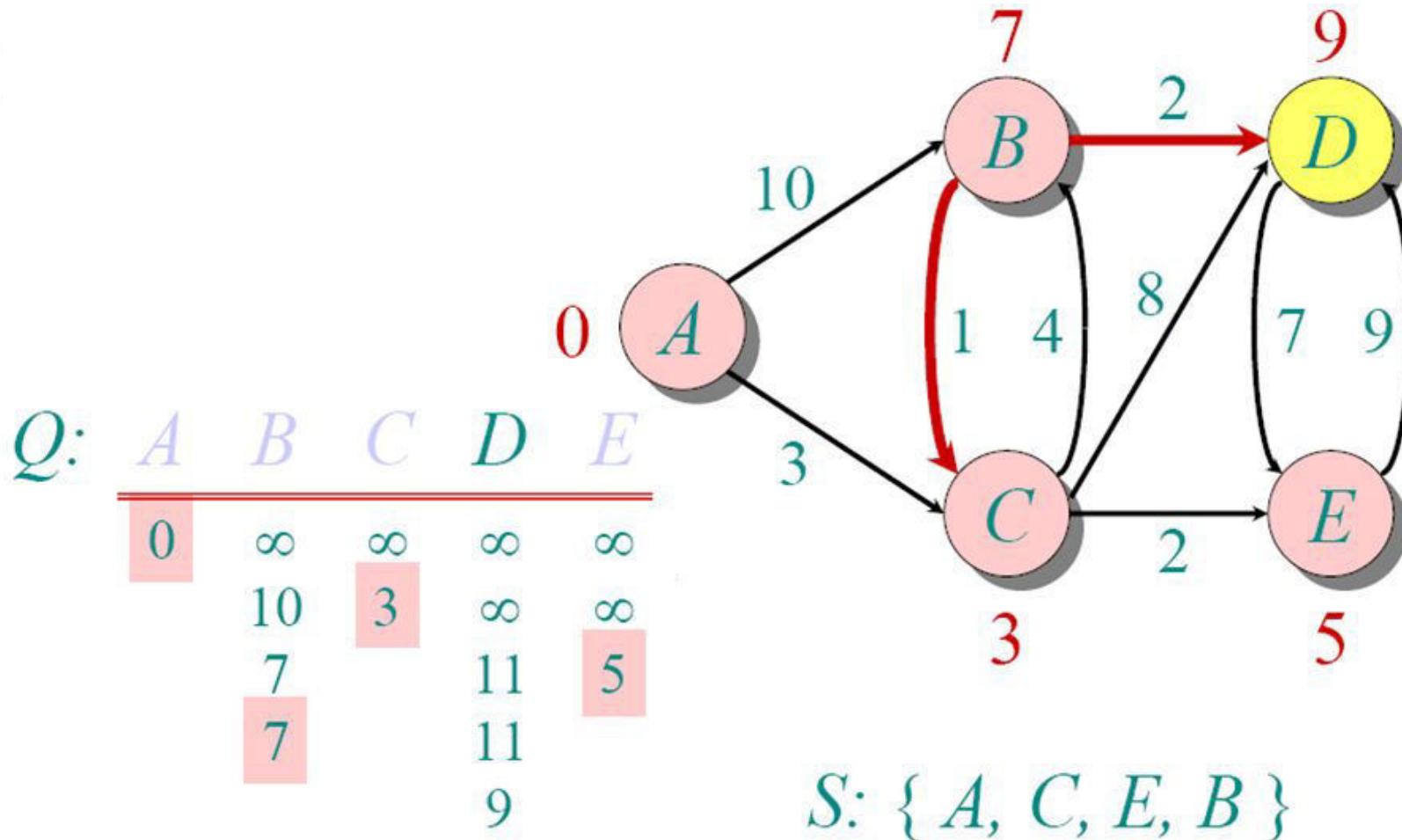
Another Example



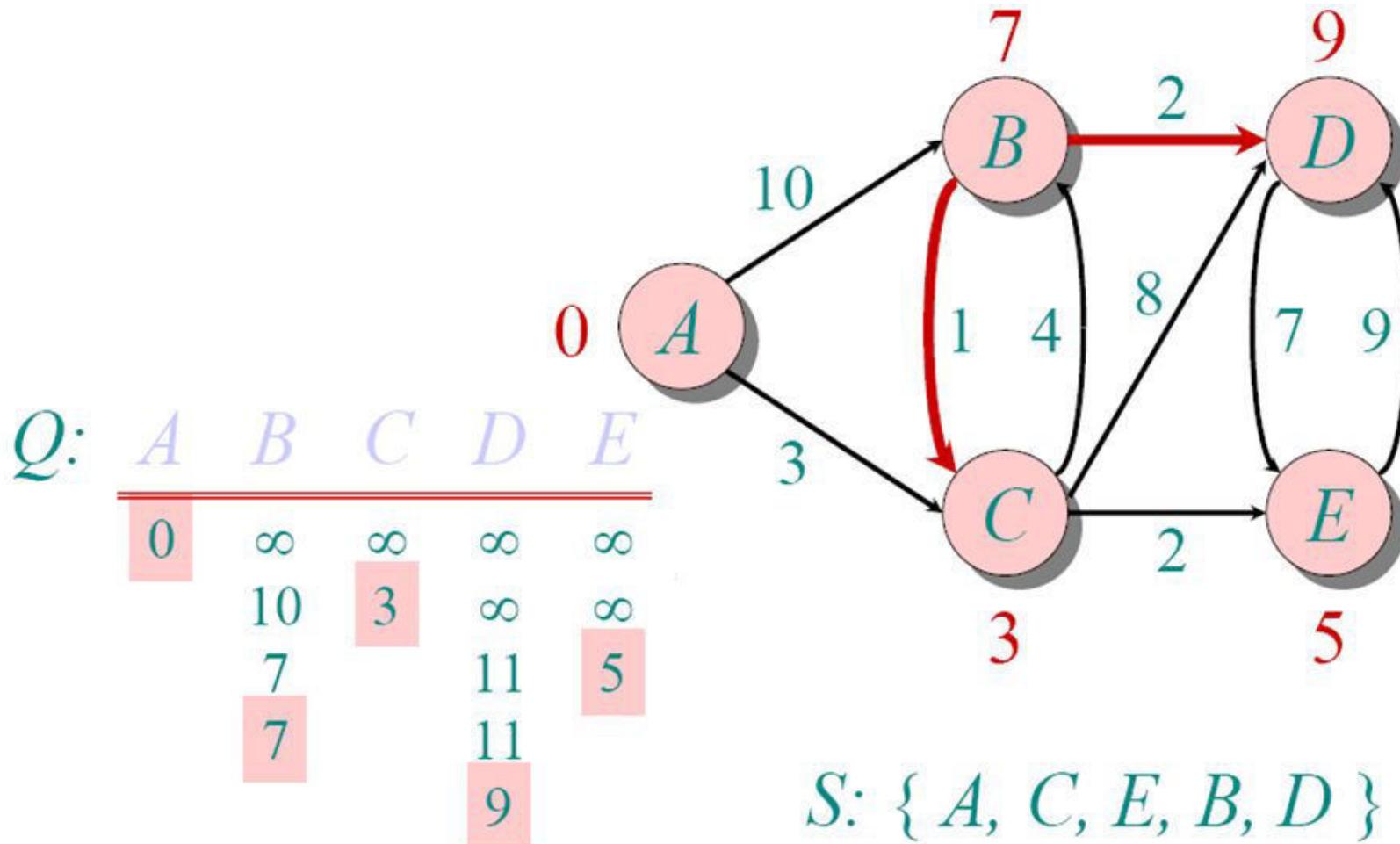
Another Example



Another Example



Another Example



Dijkstra's Pseudo Code

- Graph G , weight function w , root s

```
DIJKSTRA( $G, w, s$ )
1 for each  $v \in V$ 
2     do  $d[v] \leftarrow \infty$ 
3  $d[s] \leftarrow 0$ 
4  $S \leftarrow \emptyset$      $\triangleright$  Set of discovered nodes
5  $Q \leftarrow V$ 
6 while  $Q \neq \emptyset$ 
7     do  $u \leftarrow \text{EXTRACT-MIN}(Q)$ 
8      $S \leftarrow S \cup \{u\}$ 
9     for each  $v \in \text{Adj}[u]$ 
10        do if  $d[v] > d[u] + w(u, v)$ 
11            then  $d[v] \leftarrow d[u] + w(u, v)$ 
```

relaxing
edges

Time Complexity: Using List

The simplest implementation of the Dijkstra's algorithm stores vertices in an ordinary linked list or array

- Good for dense graphs (many edges)
- $|V|$ vertices and $|E|$ edges
- Initialization $O(|V|)$
- While loop $O(|V|)$
 - Find and remove min distance vertices $O(|V|)$
- Potentially $|E|$ updates
 - Update costs $O(1)$

Total time $O(|V^2| + |E|) = O(|V^2|)$

Time Complexity: Priority Queue

For sparse graphs, (i.e. graphs with much less than $|V|^2$ edges)

Dijkstra's implemented more efficiently by *priority queue*

- Initialization $O(|V|)$ using $O(|V|)$ buildHeap
- While loop $O(|V|)$
 - Find and remove min distance vertices $O(\log |V|)$ using $O(\log |V|)$ deleteMin
- Potentially $|E|$ updates
 - Update costs $O(\log |V|)$ using decreaseKey

Total time $O(|V|\log|V| + |E|\log|V|) = O(|E|\log|V|)$

- $|V| = O(|E|)$ assuming a connected graph

Discussion Questions

- A. Explain what happens in the following code.

```
node* egg = NULL;           // why set the initial list pointer to NULL?  
list_head_insert(3, egg);  
list_head_insert(5, egg);    // what's the egg list look like now?  
node* cheese = NULL;  
list_head_insert(9, cheese);  
list_head_insert(13, cheese); // what's the cheese list look like now?  
cheese = egg;              // what's the cheese list look like now?  
list_clear(egg);           // what's the cheese list look like now?
```

- B. Notice that one of the list operators is a copy method. Why would we NOT overload the assignment operator = for nodes to copy a list to another list in this case? A hint: what would it do to the linked list code in review question #3, where you say

```
node* cursor;  
cursor = head_ptr;
```

if an assignment operator existed? Explain whether this might pose a problem.

- C. Write a non-member function that takes 2 linked lists as constant arguments, and returns a new linked list that contains every number the two lists have in common. One or both of the input lists may be empty, or may contain numbers. Don't change either of the two input lists. Assume that each list has at most only one copy of any particular number (so that the list 3, 5, 6, 7, 3, which has 2 copies of 3, is not allowed for this question).

```
node* intersection(const node* head1, const node* head2)  
{  
}
```

If list1 contains the numbers 3, 2, 4, 9, 8, 1, 0, 5 and list2 contains the numbers 0, 1, 7, 6, 4, 5, 9, then

```
node* intersect_list;  
intersect_list = intersection(list1, list2);
```

should return a list of the numbers 0, 1, 4, 5, 9 in any order.

- D. Write a non-member function that takes 2 linked lists as constant arguments, and returns a new linked list that contains every number the two lists do NOT have in common. Use the same assumptions as in part C.

```
node* disjunction(const node* head1, const node* head2)  
{  
}
```

If list1 contains the numbers 3, 2, 4, 9, 8, 1, 0, 5 and list2 contains the numbers 0, 1, 7, 6, 4, 5, 9, then

```
node* disjunct_list;  
disjunct_list = disjunction(list1, list2);
```

should return a list of the numbers 2, 3, 6, 7, 8 in any order.

E. What's the order of the operations described in parts D and E? If the lists were sorted, would this order change? Explain. Rewrite the routines in D and E assuming that list1 and list2 are already sorted smallest to largest.

F. Suppose we have a doubly linked list: 3, 1, 8, 5, 4, 2, 9, 6, 7, 0. Write a routine that swaps any two nodes in the list, resetting their forward and back pointers as needed. You should of course check that nodes a and b are in the list (however you like) and then exchange them in the list if both are present. Notice that the head pointer may change as a result of this operation. And remember to pay attention to the special cases, like:

a and b are separated by one node in the list,

a and b are adjacent to each other in the list,

a or b is the head or the tail of the list.

```
void swap(const node*& head_ptr, node*&a, node *& b)
{
    // up to you
}
```

1. A Stack is a _____ DS
 - a. LIFO
 - b. FIFO
 - c. LILO
 - d. FILO
2. In a Stack the elements are entered from _____.
 - a. End
 - b. Top
 - c. Random
 - d. Bottom
3. The basic Operations on the Stack are
 - a. Push & Pop
 - b. Insert & Delete
 - c. Enter & Remove
 - d. Add & Subtract
4. Condition Checked before Pushing
 - a. Stack Overflow
 - b. Stack Underflow
 - c. No Condition to be checked
 - d. Both the conditions
5. Condition Checked before Poping
 - a. Stack Overflow
 - b. Stack Underflow
 - c. No Condition to be checked
 - d. Both the conditions
6. The total nos of pair of brackets in an expression is called _____
 - a. Open Scopes
 - b. To be closed Scopes
 - c. Nesting Depth
 - d. Depth Sequence
- Stacks are implemented using _____.
 - a. Arrays
 - b. Linked Lists
 - c. Static Variables
 - d. Register Variables
8. Recursive function call uses _____ Data Structure.
 - a. Stacks
 - b. Queues
 - c. Linked Lists
 - d. Trees
9. Entries in a stack are "ordered". What is the meaning of this statement?
 - a. A collection of stacks can be sorted.
 - b. Stack entries may be compared with the '<' operation.
 - c. The entries must be stored in a linked list.
 - d. There is a first entry, a second entry, and so on.

10. The operation for adding an entry to a stack is traditionally called:

a. add b. append c. insert d. push

11. In the linked-list stack class, which operations require linear time for their worst-case?

a. is_empty b. peek c. pop d. push

12. What is the value of the postfix expression $6\ 3\ 2\ 4\ +\ -\ *:$

- a. Something between -15 and -100 b. Something between -5 and -15
c. Something between 5 and -5 d. Something between 5 and 15

13. Here is an infix expression: $4+3*(6*3-12)$. Suppose that we are using the usual stack algorithm to convert the expression from infix to postfix notation. What is the maximum number of symbols that will appear on the stack AT ONE TIME during the conversion of this expression?

- a. 1 b. 2 c. 3 d. 4 e. 5

14. The operation on stack that increments the top is called

- a. Overflow b. Push c. Pop d. Underflow

15. The situation of deleting an element from a stack which doesn't have elements is called

a. Overflow b. Push c. Pop d. Underflow

16. The time complexity of adding an element to a stack of n elements is?

- a. O(1) b. O(n) c. O(n^*n) d. None

17. The time complexity of deleting an element from a stack of n elements is?

- a. O($n+1$) b. O(n) c. O(1) d. None

18. A stack is said to be Full when it is _____

- a. Unsorted b. Underflow c. Over flow d. Sorted

19. B(i) represents the bottom of the stack i & T(i) represents the top stack. When stack will become full?

- a. B(i)=T(i) b. B(i+1) = T(i)+1 c. B(i-1) = T(i+1) d. B(i-1) = T(i-1)

20. What type of storage is used to represent stacks and queues
- a. Random b. Sequential c. Dynamic d. Logical
21. The operation for removing an entry from a stack is traditionally called:
- a. delete b. peek c. pop d. remove
22. Which of the following stack operations could result in stack underflow?
- a. Is empty b. Continuously popping c. Continuously pushing d. This can never happen
23. Which of the following applications may use a stack?
- a. A parentheses balancing program. b. Keeping track of local variables at run time.
- c. Syntax analyzer for a compiler. d. All of the above.
24. Consider the following pseudo code:
- declare a stack of characters
- while (there are more characters in the word to read)
- { read a character
- push the character on the stack
- }
- while (the stack is not empty)
- { write the stack's top character to the screen
- pop a character off the stack
- }

What is written to the screen for the input "carpets"?

- a. serc b. carpets c. steprac d. ccaarrppeeettss

25. Here is an INCORRECT pseudo code for the algorithm which is supposed to determine whether a sequence of parentheses is balanced:

```
declare a character stack  
  
while ( more input is available)  
  
{ read a character  
  
if ( the character is a '(' )  
  
push it on the stack  
  
else if ( the character is a ')' and the stack is not empty )  
  
pop a character off the stack  
  
else print "unbalanced" and exit  
  
}  
  
print "balanced"
```

Which of these unbalanced sequences does the above code think is balanced?

- a. ((()) b. ())(() c. (()())) d. (()))()

26. Consider the usual algorithm for determining whether a sequence of parentheses is balanced. What is the maximum number of parentheses that will appear on the stack AT ANY ONE TIME when the algorithm analyzes: ((())(())?)

- a. 1 b. 2 c. 3 d. 4 e. 5 or more

27. Suppose we have an array implementation of the stack, with ten items in the stack stored at data[0] through data[9]. The CAPACITY is 42. Where does the push member function place the new entry in the array?

- a. data[0] b. data[1] c. data[9] d. data[10]

28. Consider the implementation of the stack using a partially-filled array. What goes wrong if we try to store the top of the stack at location [0] and the bottom of the stack at the last used position of the array?

- a. Both peek and pop would require linear time.
- b. Both push and pop would require linear time.
- c. The stack could not be used to check balanced parentheses.
- d. The stack could not be used to evaluate postfix expressions.

29. In the linked list implementation of the stack, where does the push member function place the new entry on the linked list?

- a. At the head
- b. At the tail
- c. After all other entries those are greater than the new entry.
- d. After all other entries those are smaller than the new entry.

30. In the array version of the stack (with a fixed-sized array), which operations require linear time for their worst-case behavior?

- a. is_empty
- b. peek
- c. pop
- d. push